测试馒头铺从0到1职业规划丛书

Python接口自动化测试

王浩然◎著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书没有采用传统的教科书写作模式，而是从要实现的目标着手，将“Python+MySQL 处理 HTTP 接口”过程拆分成一个个知识点，最后串联各个知识点。本书主要介绍了如何用 Python 实现接口自动化测试。全书主要内容包括接口基础、接口手工测试、编程前的准备、用 Python 操作 MySQL 数据库、用 Python 发送 HTTP 请求、用 Python 处理 HTTP 返回包、用 Python 导出测试数据、接口自动化及实际接口场景演示。

本书适合初、中级测试工程师，对 Python 语言感兴趣的人员，以及想要提升技术的人员。

图书在版编目（CIP）数据

Python 接口自动化测试 / 王浩然著. —北京：电子工业出版社，2019.5
（测试馒头铺从 0 到 1 职业规划丛书）
ISBN 978-7-121-35687-2

Ⅰ. ①P… Ⅱ. ①王… Ⅲ. ①软件工具－自动检测Ⅳ. ①TP311.561

中国版本图书馆 CIP 数据核字(2018)第 280896 号

策划编辑：王　静
责任编辑：牛　勇
印　　刷：
装　　订：
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×980　1/16　印张：14.5　字数：230 千字
版　　次：2019 年 5 月第 1 版
印　　次：2019 年 5 月第 1 次印刷
定　　价：59.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

序

如今，软件测试岗位受到很大的挑战。一方面，敏捷开发模式、DevOps 实践等愈发流行，其强调开发与测试相融合，即测试人员能干开发的工作，开发人员能干测试的工作；另外一方面，测试行业开始认为自动化测试很重要，如果实现不了自动化测试，那么测试就无法敏捷起来（特别是在快速迭代、持续交付的环境下）。

说起自动化测试，根据公众号“软件质量报道”和相关机构最近的调查，目前的自动化测试（特别是面向 GUI 的自动化测试）效果还不够好，产出投入比不高，自动化测试做得好的公司或团队也不多。但是，基于 API 进行自动化测试（接口自动化测试）还是比较容易实施的，自动化率能达到 90%以上，并且投入产出比高。另外，如今软件架构也慢慢转向 SOA 架构、微服务架构，基于 API 进行测试的需求越来越大，这给自动化测试提供了更多的机会。

本书正是帮助那些自动化测试基础比较弱，甚至是零基础的测试工程师转型做接口自动化测试，而且是基于现在如日中天的 Python 语言来开发自动化脚本，对渴望入门 Python 编程的朋友也有价值。本书循序渐进地引导读者完成接口自动化测试。

本书直接基于 Python 代码来实现接口自动化测试，不依赖其他测试工具，降低了学习门槛和使用成本。自动化测试所需的 Python 技能其实也很简单，读者也不用恐惧，用

一些资深人士的话说：1~2 天就可以了。即使学得慢一些，一周就能学会。在接口自动化测试过程中，一般建议以自动生成测试数据为主（即先基于自定义的业务数据模板来自动生成大部分测试数据，然后手工再补充一些特殊的测试数据）。未来可以结合人工智能算法来完善测试数据、自动分析与生成接口调用链等，虽然本书没有介绍这方面的内容，但有了本书作为的基础，读者就可以通过自学深入下去。

最后需要提醒读者：想要做好测试，深刻理解用户、产品和业务是非常重要的。任何商业软件最终都是为了解决业务问题和满足用户的需求，而测试正是对这种质量的保障。所以，在学习自动化测试的过程中，一定要重视业务需求、测试思维和测试方法等。只有具备良好的测试素质，才能让自动化测试发挥其价值，才能真正做到事半功倍。

朱少民

国内知名测试专家

前言

本书整体设计思想

自动化测试的前景

软件测试，在大多数的公司中还是处于相对弱势的地位，原因主要在软件测试本身：相比于软件开发，软件测试无论进入门槛还是编程能力，要求都低一些；而且大多数公司的软件测试还局限于手工测试。这就造成了业界对软件测试的偏见——软件测试只是随便按一按鼠标，技术含量低，比不得架构设计和模块化编程等。

诚然，软件测试离不开手工测试，但不能因手工测试而忽视有编程能力。技术性测试的方法有很多，常见的有自动化测试、性能测试、白盒测试、安全性测试等，这些测试方法都需要测试人员有比较强的编程能力。其中，自动化测试的进入门槛较低，但效果最明显，所以，自动化测试可以作为广大测试人员进入技术性测试的切入点。

现阶段，薪资高一些的测试岗位，普遍要求从业人员具有自动化基础及实际操作能力。所以，从就业角度来说，自动化测试是突破测试行业薪资瓶颈的一条捷径；从测试人员的职业发展来说，掌握测试编程技术，有助于建立技术思维及行业内部的沟通，便

于将来继续走测试技术路线，或是转到其他岗位。

本书写作目的

相信很多测试人员和我的经历很像，从最开始的手工测试开始积累经验，在这个过程中肯定想过做自动化测试、做技术。我最开始是用 QTP（Quick Test Professional，一种自动测试工具）录制了登录功能，看着浏览器自动打开、自动输入网址、自动登录，那种愉悦的心情是不言而喻的。但是，真正华丽转型（或者说能有一技之长）的测试人员还是偏少的，原因就在于难以突破关键技术点。

各种编程语言本身都有相同点，只要掌握了一门，其他语言学起来也就没那么难了。所以，如何掌握一门语言，如何跨过第一道门槛，成为制约测试人员技术能力提升的关键点，这也是本书编写的目的——引导测试人员突破 Python 的入门难点。

为什么要选择 Python？最主要的原因是——合适。正所谓“鞋合不合适，只有脚知道”。Python 简单易学的特点，恰好符合测试人员的要求。很多时候，我都在怀疑 Python 是不是专为测试人员量身打造的。关于 Python 的更多优势，读者可以在正文中看到。

本书特点

作者在构思本书的时候，也翻阅了市面上很多同类技术书籍，发现大多数都有一个通病——采用教科书的写作模式。所谓“教科书模式”，即按照“语言的历史→语法→章节练习→案例讲解”的模式来展开。这种模式很经典，内容也很翔实，但是太过于死板和理论化。这样的书虽然是结构明晰、循序渐进，仔细阅读后确实能收获很多，但是不利于快速上手。

读教科书模式的书时，学习者很容易半途而废，包括我自己，经常是看了不到三分之一的内容就放弃了，所有的新鲜劲都在语法和练习中被消磨殆尽了。我真正开始写脚本并不是从某本书开始的，而是从实际的项目拆分入手的，遇到问题再去翻书，纯粹将

它当作工具书来使用。

所以，本书不是按照传统的教科书模式编写的，而是从实际要实现的目标着手，一步步将目标拆分成知识点，再对知识点进行细分，将每个点拆分成一个个小的突破点——类似于“拆书帮”（某个学习社区）的形式，将“Python+MySQL 处理 HTTP 协议接口的过程”拆解到一个个章节中。读者在每个章节中都能学到完整的一个知识点，最后串联各个知识点，实现最终的学习目标。关于如何分解、如何逐个突破，读者可以在正文中看到。

读者通过实现每个章节的功能，逐步加深对 Python 的理解，通过小篇幅的功能实现来提升成就感，激发自己继续往下看、继续往下学的信心和勇气。本书所介绍的方法是作者在实际项目中实践过的，并且也被很多业内同行所采用。读者不仅可以将其用在 Python 学习中，也可以用在其他语言甚至生活中。

读者对象

- 对 Python 感兴趣的人员；
- 想在项目中实现 HTTP 协议接口自动化测试的人员；
- 想要提升技术的人员；
- 初、中级测试工程师。

代码下载

本书配套代码的下载地址：https://www.broadview.com.cn/35687。

有关于任何问题、建议和疑问，欢迎发邮件到：smallprocess@yeah.net。

致谢

感谢 Guido van Rossum 于 1989 年发明了 Python。

感谢电子工业出版社出版此书，以及为本书能够快速出版而做的审校、加工等辛苦工作。

感谢何飞在本书构思和出版中所做的指导和帮助。没有他的指引就没有本书的出现。

感谢我的家人，感谢你们对我工作的理解和支持，有你们一直的付出才有今天的这本书。

作　　者

目录

1 本书整体设计思想

本章讲什么：

（1）为什么要做懂技术的测试人员；

（2）为什么选择这本书，为什么选择 Python；

（3）本书能给你带来什么；

（4）自动化代码的设计思路。

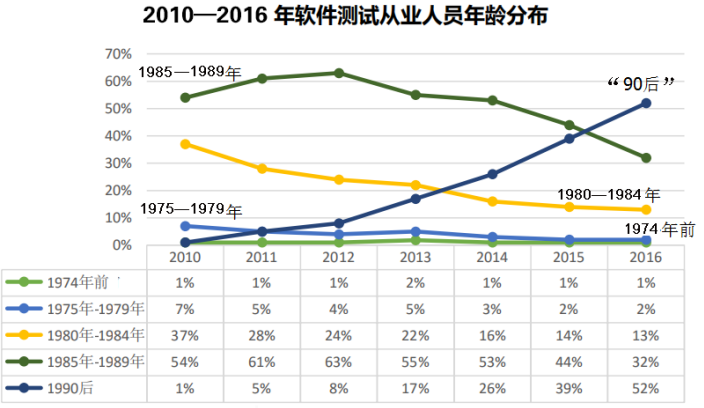
本章主要介绍如何构思和实现接口自动化测试。传统的教学模式固然有其突出的一面，但如何能让读者快速入门、快速上手、快速产出才是本书的出发点。所以，本书的总体思想是“学以致用”，以目的反推要做的每一步，逐步实现每一步，需要什么就学什么，目的性很明确，最后再将“已实现的每一步” 串联，达到最终的效果。

1.1 为什么要做懂技术的测试人员

物竞天择，适者生存。

相当一部分人选择软件测试（以下简称测试）行业的原因是，测试相对开发来说，其技术门槛更低。在 IT 行业，测试岗位对编程能力的要求相对较低。直白地说，只要会操作电脑就可以做测试。最重要的是，测试人员的待遇相对其他岗位来说还是比较高的，工作也是“白领”模式。所以，大批的新鲜血液源源不断地汇入测试行业。这也造成了在测试行业中，技术含量比较低、纯手工的重复性工作偏多。在笔者之前所在的公司中，测试部门居然不属于技术部门，可见大环境给测试工作贴的“标签”就是“技术含量较低”。

图 1-1-1 所示的是 51Testing 发布的 2010—2016 年软件测试行业人员年龄分布图。很明显，“90 后”已经成为测试行业的主力军，而“80 后”拿什么和这些年轻人竞争呢？



	2010	2011	2012	2013	2014	2015	2016
1974年前	1%	1%	1%	2%	1%	1%	1%
1975年-1979年	7%	5%	4%	5%	3%	2%	2%
1980年-1984年	37%	28%	24%	22%	16%	14%	13%
1985年-1989年	54%	61%	63%	55%	53%	44%	32%
1990后	1%	5%	8%	17%	26%	39%	52%

图 1-1-1　2010—2016 年软件测试行业人员年龄分布图

说到这里，笔者有几个问题想问工作 5 年左右的黑盒测试人员：

（1）你觉得自己比工作 1~3 年的测试人员强在哪里？

（2）如果给你的工资可以雇佣 1.5 个工作了 1~3 年的黑盒测试人员，而且他们更年轻、能加班，那么公司为什么要选择你呢？

（3）工作 5 年左右的测试人员一般都成家生子了，那么你怎么保证自己的工作效率和产出会比工作 1~3 年的黑盒测试人员高呢？你怎么保证能在家庭和工作之间找到平衡点？

可能会有人愤愤不平，觉得自己有 5 年的工作经验。然而，在一个行业中，工作经验的积累在前 3 年是呈指数型增长的，而在 3 年后则是趋于平稳的，即 5 年工作经验和 3 年工作经验其实相差无几。在这里，我用 5 年工作经验来举例，并不是说对 5 年工作测试经验有什么偏见，而是因为，5 年恰恰是最吃香的工作年限，论经验有经验，论技术也应该有一定的积累了，所以 5 年是一个坎儿，是一个测试人员在人生道路上面临的分岔口。之前听过一个笑话：有个人去面试，说自己有 3 年工作经验。面试官问他：“你才毕业 1 年，怎么有 3 年工作经验，那两年是哪里来的？”答：“加班……”所以说，工作年限不一定能作为衡量一个人价值的标准。那么问题来了，什么才是体现你年限优势的资本呢？技术！技术！技术！重要的事情说 3 遍。

（1）纯粹的手工测试越来越趋于低门槛，人员也越来越年轻化。年纪大的测试人员不可能也不能去和这些年轻人比时间、比耐力。长江后浪推前浪，如果不想做“被拍死在沙滩上的前浪”，就要脱离这片纯手工测试的“苦海”。你可以把手工测试作为入行的首选技能，但不能只有这一项技能。所谓“技多不压身”，技术可以用来辅助测试、提高手工测试的效率。

（2）懂技术的测试人员越来越受市场欢迎。市场是检验一切能力的“试金石”，从现实来看，市场需要什么技能，你就应该掌握什么技能，只有这样才能一直被市场所追捧，才能自信地来一次“说走就走的旅行”。图 1-1-2 所示的是 51Testing 发布的 2016 年软件测试从业人员从事的测试工作类型。物以稀为贵，越少人涉及的领域越是稀缺的

领域，也越是市场追捧的领域。相对手工测试，白盒测试、安全性测试涉及的人数少之又少，为什么？主要是因为这两种测试工作都需要有强大的技术支持，所以也是市场回报率极高的工作。所以，做一个有技术的测试人员吧！市场是不会亏待手艺人的。

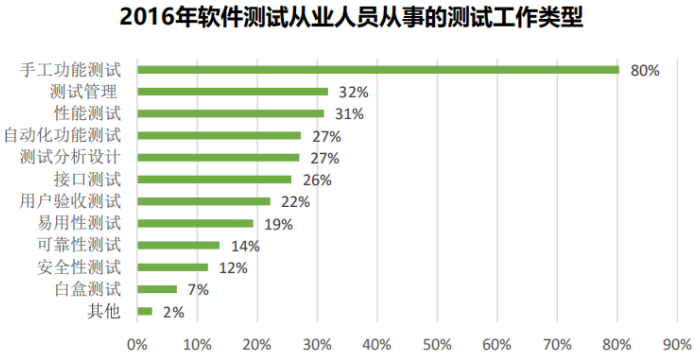


图 1-1-2　2016 年软件测试从业人员从事的测试工作类型

（3）懂技术的测试人员更容易和开发人员沟通。开发人员和测试人员，一直被认为是一对“爱恨交织”的组合。开发人员觉得测试人员没有技术含量，测试人员认为开发人员只会编写代码，情商低得吓人。《孙子兵法》有云：“知己知彼，百战不殆。”想要和开发人员很好地交流，首先要进入他们的世界。一个有技术能力的测试人员，在开发人员眼中的地位是不一样的。比如，在发现 Bug 的同时提出代码错误和解决方案，会更有成就感。而且，你会发现开发人员看你的眼神、待你的态度会大有改观，因为你是一个和他有共同语言的测试人员。

（4）更长远地看，难道你想在而立之年还去应聘纯手工测试岗位？还要和一群初出茅庐的新人竞争功能测试岗位吗？难道你不想因为一技之长而被市场所追捧吗？难道你不想和开发人员站在“平等”的地位看问题吗？难道你不想和开发人员有更多的共同语言吗？对测试人员来说，技术不是万能的，但是没有技术是万万不能的！

1.2　为什么选择这本书

市场上关于 Python 测试的书多如牛毛，在网上这类文章也数不胜数，为什么读者

还要花钱购买这本书呢？

（1）市场上的这些图书，基本都是先讲语法，再讲代码实例，最后再放几个练习题。此时读者的耐心在语法环节已经被消磨得差不多，再去看代码还能记住多少语法？有鉴于此，本书颠覆了传统的编写模式，以编写一套可落地的接口自动化代码为目标，将目标拆分为要实现的单个步骤，再将每个步骤合并成模块，然后对模块进行编码实现，最后将模块串联起来。在这个过程中，读者可以同步学习每个模块所涉及的 Python 语法，将语法融入代码中，真正做到学以致用。

（2）本书在“千聊”平台有同步课程，该课程已服务一大批有志的测试工程师。他们中的很多人都是零基础，在认真学习完课程后，都能实际上手接口自动化测试。本书是对课程的提炼和补充，是实实在在的一线测试干货积累。有了这本书，就等于有了一套实实在在的测试平台。

1.3 为什么选择 Python

有句话说得好：“适合自己的才是最好的！”Python 适合测试人员的原因如下：

（1）学习难度小，开发周期短。目前国内大多数测试人员往往编码经验不足，Python 是一门很好的入门语言。

（2）具有“胶水语言”特性，能与 C++、Java、.NET、Object-C 等语言整合。

（3）语法简约、清晰，模块资源丰富。

（4）跨平台。

（5）有很多成熟的框架，如 Django、Twisted 等。

（6）可移植性强。

Python 对测试人员来说，是性价比极高的语言，是他们升职加薪的必备技能。

当然，Python 不仅仅用来辅助测试，还能做很多其他你想不到的事情，比如：

（1）从入门级开发者到专业级开发者都在做的——爬虫。

（2）开发 Web 程序。

（3）开发桌面程序。

（4）大数据分析与计算。

（5）图像处理。

（6）实现人工智能。

（7）实现安全。

1.4 本书能给你带来什么

“无法落地的技术、理论，都是耍流氓”，本书承诺不做“流氓”。

（1）本书提供落地的 HTTP 接口自动化代码，提供实实在在的可运行环境。很多测试工程师学习自动化测试和 Python 的初衷都是希望能开发一套能使用的代码。本书提供了一套完整的自动化测试代码，读者在这个基础上可以优化、拓展、自由发挥。

（2）本书只讲 Python 的入门语法，不讲“高大上”“难偏全”的语法。本书涉及的语法都是代码中用到的，并在这个基础上做适当的延伸，让读者在编写代码的同时知其所以然。

（3）源码公开。拥有这本书，就拥有了一整套的源码。只要按照书中所介绍的环境配置步骤操作，源码完全可运行。

1.5 自动化代码的设计思路

在介绍具体设计思路之前，给大家举一个例子：把大象放进冰箱，总共分为几步？

先不管冰箱有多大、大象怎么可能放得进冰箱这些实际问题，按照正常的步骤应该

分为三步：

打开冰箱门 → 把大象放进去 → 关上冰箱门

举这个例子是要告诉大家，要实现某个目标，先不要考虑实现的具体细节和效率、实现的道路有多曲折。因为在你面前压根儿还没有路，与其思来想去，不如找准方向，勇敢地踏出第一步。按照能想到的步骤进行拆分，然后逐个突破，而这个过程会逐步拓展你的思路和带来成就感，这对整体的进度是有促进作用的。随着对各个点的突破，思路会越来越清晰，动力也会越来越足，自然就能实现最终的目标。

所以，可以按照图 1-5-1 所示的思路来设计主要的突破点。

图 1-5-1　实现目标的步骤

第一步：以结果为导向，反推要实现这些结果总共分为哪些步骤。比如，目标是“用 Python+MySQL 实现 HTTP 接口自动化测试”，那以这个目标为基础，反推要实现这个目标需要实现哪些功能，比如数据存储在哪里、怎么读写数据、怎么发请求、怎么处理请求等。

第二步：将每一步抽象为功能点。即将每一步抽象出来的每一个待解决的问题提炼成一个个功能点。比如，数据存储在哪里可以提炼为数据库功能。如果选择 MySQL 作为数据库，则涉及 MySQL 数据库本身的操作、SQL 语句等。这些都是由待解决问题提炼出来的功能点。

第三步：各个击破，逐一解决每一个功能点，然后再将功能点串联起来。这涉及两个过程：第一个过程是，逐一实现每个功能点，这个相对简单，即使不会，通过互联网搜索也能找到解决方案；第二个过程是，首先将功能点串联起来，在每个功能点都实现之后，你心中已经有了大概的处理逻辑，然后绘制业务流程图将每个业务功能串联起来，查漏补缺。当业务流程中所有的功能点都实现时，目的也就达到了。

1.5.1 由手工测试分析出哪些步骤可自动化处理

手工测试接口的步骤如下所示。

在 Excel 中准备测试用例→手工获取测试数据→使用 Fiddler 发送接口请求→人工比较请求结果→请求成功：更新测试用例，具体流程如图 1-5-2 所示。

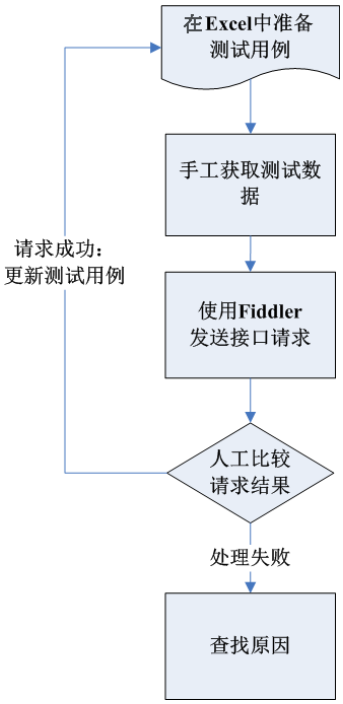


图 1-5-2 手工测试接口流程

引入自动化测试是为了解决重复性的工作。下面分析这个流程中哪些步骤是重复的：

（1）手工获取测试数据，每次都要人工复制一次数据（绝对重复）。

（2）使用 Fiddler 发送接口请求，每一条测试用例都要调用一次接口（绝对重复）。

（3）人工比较请求结果（部分重复）。

1.5.2　以可重复步骤为契机，梳理自动化测试的步骤

将重复的步骤实现机器化，则自动化测试的步骤如下。

（1）用存储设备存储测试数据，比如用 MySQL 数据库存储接口测试用例表。

（2）使用 Python 语言从存储设备中获取测试数据。

（3）使用 Python 语言发送接口请求，数据就是第（2）步中获得的测试数据。

（4）使用 Python 语言处理接口比较返回结果和预期结果。

（5）使用 Python 语言将处理结果写入对应的存储设备中。

（6）使用 Python 语言分析测试结果，以图表化的方式展示。

上述步骤如图 1-5-3 所示。

图 1-5-3　接口自动化测试功能拆分

1.5.3 抽象自动化步骤到功能点

（1）使用 MySQL 数据库存储数据。建立接口测试用例表，用来存储接口测试所需要的测试数据，要求表中的每一条数据都可独立运行。

（2）使用 Python 读写 MySQL 数据库操作，该操作是为获取测试用例数据并将结果写入对应用例做准备。

（3）使用 Python 发送 HTTP 接口请求，需要注意请求类型的不同和数据的处理逻辑。

（4）使用 Python 处理 HTTP 返回包数据，这里是从两个维度（关键参数值与参数完整性）来进行处理。

（5）使用 Python 统计、导出测试结果，涉及 Excel 操作和数据的分析和图表化。

（6）梳理业务处理逻辑，整合功能点，实现最终效果。

将上述步骤简化后如图 1-5-4 所示。

图 1-5-4　接口自动化功能抽象

1.6 补充知识点

1.6.1 什么是面向对象编程中的对象

动物园里有各种各样的动物，每一个动物都可以被称为一个“对象”。动物园在管

理动物时，会按照大类将动物分别圈养，而这个“大类”就是在每个对象身上寻找共同点，有共同点的放在一起。

同理，在面向对象编程中，所有的数据类型都被称作“对象”，将有共同点的对象抽象出来就变成了“类”。

综上所述，对象是 Python 中最基本的单位，所有的事物都可被称为“对象”。

1.6.2 什么是面向对象编程中的类

“物以类聚，人以群分”是指，同类的东西常聚在一起，志同道合的人相聚成群。我们经常说这类人（比如精明的人）如何如何，“这类人”应该是有共同点的。如果将这个概念延伸到编程语言的类中，那“类”应该是具有共同点的一群事物，比如人类、鱼类、鸟类等。

这些共同点是什么呢？现实中说“这类人”（精明的人），一般是指这类人聪明、会算计等，这些就是这类人的共用特性。同样，将这个概念延伸到编程语言的类中，那么“类”应该有一些静态的和动态的属性。比如人类，静态的属性有一个脑袋、有手、有脚、会呼吸等；动态的属性有直立行走、会制造工具等。

综上所述，编程语言中的“类”就是有特定属性（静态、动态）的一个基本组合。

1.6.3 什么是编程语言中的实例

“实例”总是和“类”绑定在一起使用的。即实例是依附于类存在的，实例是类的代言人。可以这么举例，“类”相当于古时候的皇帝，皇帝想知道下面的官员是否按照自己的旨意去办事，但皇帝（类）又不能亲自出马，所以委派了钦差大臣（实例），让他行使皇帝赋予的权利，这时钦差大臣就代表皇帝，拥有皇帝才有的权利。

“实例”代表了“类”，但实例能被使用，正是使用了实例的方法。所以，使用实例达到了使用类的效果。

1.6.4 自动化测试是不是比手工测试覆盖率高

1．分析

有的读者可能觉得自动化很神奇，认为自动化测试能替代手工测试。然而，“理想很丰满，现实很骨感”。

许多自动化测试最终以失败告终，究其原因，不是技术不够成熟，也不是测试人员不用心，而是因为其太想自动化了，太想把所有事都用机器去实现，到最后为了自动化而自动化，疲于应对纷繁变化的业务，疲于维护代码，最终发现实际的工作量比手工测试还要多，这样自动化项目自然就无法被推进下去了。

2．看点

自动化测试是让机器去执行代码，自动化代码是编程语言的集合，编程语言是开发者思想的体现，开发者思想是业务逻辑抽象的结果，所以“自动化=业务逻辑的抽象”。这么说读者应该能理解“指望自动化测试提高测试覆盖率是不现实的”。因为它只是业务逻辑的一种体现方式，最多只是效率要高一些而已。

3．反问

既然自动化测试不能提高测试覆盖率，为什么还要用自动化测试代替手工测试呢？虽说自动化测试没有提高覆盖率的优势，但其他方面的优势它还是有的，比如无差异地执行（手工测试不能保证每轮执行都一样）。

自动化测试的执行效率比手工高。但这个优势需要有一个基础——项目比较稳定。只有稳定的项目才有自动化的价值，所以，现在很多人摒弃了 UI 自动化测试（或者说，复杂的 UI 自动化测试）的原因是：UI 变化快，维护代码的成本太高，继而开始追捧接口自动化测试。因为接口是连接业务和数据的纽带，是业务逻辑的集中体现，同时接口又比较单一，其核心部件就三个——地址、入参、返回包。最重要的一点是，接口的修改基本以增量模式为主(即保持原有的逻辑不变,依据业务需要新增接口或摒弃原接口)，所以其稳定性很高。所以，接口自动化测试的性价比高。

4．追加

既然自动化测试性价比如此高，那么该做到什么层次呢？可以参照表 1-6-1 来定义自动化的检查等级。

表 1-6-1　接口自动化测试检查点等级

测试项目	意　　义	是否推荐自动化	备注
HTTP Code 检查	判断 HTTP 请求的结果，如果返回的是 200 则表示正常	是	-
Interface ReturnCode 检查	接口返回包的关键参数，是接口正确性的第一要素	是	-
Interface 结构完整性检查	返回包层级正确性，保证返回数据结构性正确	是	-
Interface 参数完整性检查	返回包所包含的参数集合，在结构完整性正确后检查，可保证必要参数的存在及其正确性	是	-
Interface 特殊参数值检查	特殊参数值检查，不宜过多，有别于 ReturnCode 检查，偏向于包数据	是	依据实际接口情况确定特殊参数
Interface 功能检查	返回包所有参数值的比对	否	和业务关联大，变动频繁

1.6.5　什么是自动化测试

广义上来讲，一切通过工具（程序）的方式来代替或辅助手工测试的行为，都可以被看作是自动化，甚至包括：① 性能测试工具（Loadrunner、Jmeter），其本质也是自动地发送请求，只是监控的指标不同而已；② 自己所写的一段代码，用于生成区间内的随机数字。

狭义上来讲，自动化测试是指：使用工具记录或编写代码的方式模拟手工测试的过程，通过回放或运行代码来执行测试用例，让机器重复处理从而代替人工对系统进行验证。

从上面两个层面不难理解，在做自动化测试时，不一定动辄就是搭建框架或平台，而要立足于实际的项目需要，从最重复的点着手，哪怕只是自动生成测试数据，也能提

高测试效率。“以提高效率来驱动自动化的项目的开展和落地”比“追求华而不实、大而全的框架性东西”更接地气。

1.6.6 什么是分层自动化测试

金字塔结构最初在 Mike Cohn 于 2009 年著作 *Succeeding with Agile: Software Development using Scrum*（《Scrum 敏捷软件开发》）的一书中被提到。在这本书中，自动化测试被定义为一种三层的金字塔形结构（如图 1-6-1 所示）。他的基本观点是：应该有更多低级别的单元测试，而不仅仅是通过用户界面运行高层“端到端”的测试。

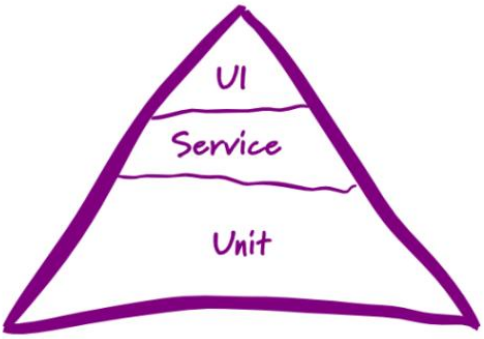


图 1-6-1 测试金字塔

传统的自动化测试，大多关注的是产品 UI（用户界面）层的自动化测试；而分层的自动化测试，倡导在产品的不同阶段都要实施自动化测试。根据测试金字塔，可设计分层自动化测试（如图 1-6-2 所示），其强调从 UI 层到最小的逻辑单元层都需要实施自动化测试，从全面自动化测试到对系统的不同层次进行不同程度的自动化测试。

相信从事测试工作的人对图 1-6-2 的金字塔图形并不陌生，这不就是在产品开发不同阶段所对应的测试吗！规范的单元测试需要相应的单元测试框架，单元测试框架有基于 Java 的 Junit、testNG，基于 C#的 NUnit，以及基于 Python 的 unittest、pytest 等。几乎所有的主流语言，都会有其对应的单元测试框架。

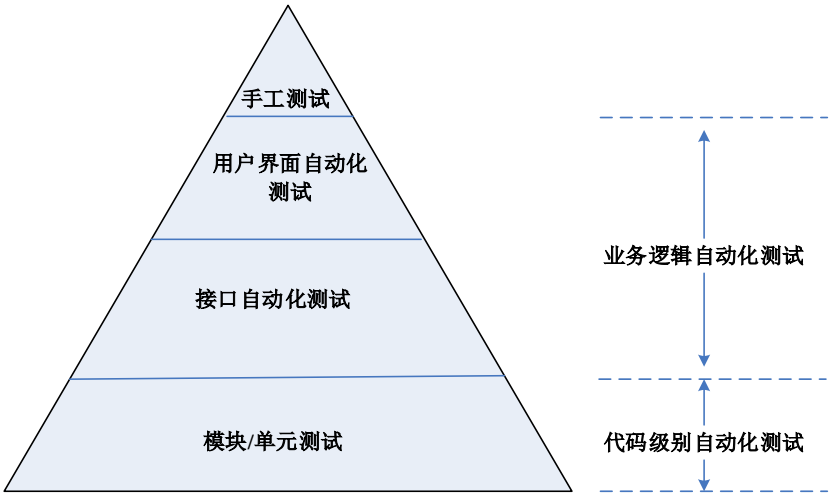


图 1-6-2　分层自动化测试

1．手工测试

对于手工测试，读者应该不陌生，它是测试入门必须经历的测试。手工测试是由人一个个地输入用例，然后观察结果，属于比较原始但是必需的一个步骤。这种测试主要应用在程序集成阶段，即在产品已具有较完善的功能时再测试。

2．用户界面自动化测试

大部分测试人员的大部分工作都是对用户界面的功能进行测试。例如，不断重复地对一个表单进行提交操作，以测试其功能。这时可以通过相应的自动化测试工具来模拟这些操作，从而避免重复劳动。

UI（用户界面）层的自动化测试工具非常多，比较主流的是 QTP、Robot Framework、Watir 和 Selenium。

为什么要画成一个金字塔形，则不是长方形或倒三角形呢？这是为了表示不同阶段所投入自动化测试的比例。

如果一个产品从没有做过单元测试与接口测试，只做 UI 层的自动化测试，是不科学

的，这很难从根本上保证产品的质量。

不要妄图实现全面的 UI 层自动化测试，那是一个劳民伤财的举动：投入了大量的人力和时间，最终获得的收益可能远远低于所支付的成本。因为 UI 层的元素会时常发生改变，所以越往上层，其维护成本越高。

所以，应该把自动化测试更多地放在单元测试与接口测试阶段。

既然 UI 层的自动化测试这么劳民伤财，那只做单元测试与接口测试好了？不可以！因为不管什么样的产品，最终呈现给用户的是 UI 层。所以，测试人员应将更多的精力放在 UI 层。也正是因为测试人员在 UI 层需要投入大量的精力，所以，有必要通过自动化的方式解放部分重复的劳动。

在自动化测试中，最怕的是变化。因为变化会直接导致测试用例运行失败，从而需要对自动化代码进行维护。如何控制失败、降低维护成本，对自动化测试的成败至关重要。反过来讲，一份永远都运行成功的自动化测试用例是没有价值的。

测试金字塔中的 3 种测试的比例，应根据实际的项目需求来划分。在《Google 测试之道》一书中说道：“对于 Google 产品，70%的投入为单元测试，20%的投入为集成和接口测试，10%的投入为 UI 层的自动化测试。”

3．接口自动化测试

不少测试新手都不太容易理解接口测试。接口测试关注的是函数和类（方法）所提供的接口是否可靠。例如，要定义一个 add()函数用于计算两个参数的结果并返回，则需要调用 add()及传参，并比较返回值是否为两个参数之和。当然，接口测试也可以 URL 形式进行传递。例如，通过 GET 方式向服务器端发送请求，则发送的内容作为 URL 的一部分传递到服务器端；Web Service 技术对外提供公共接口，需要通过 SoapUI 等工具对其进行测试。

接口自动化测试是用工具和代码来模拟请求和结果校验，以解放人力和保证持续、稳定地测试。

4．模块/单元测试

在程序中最小的组合是一个个的函数或者方法，而对函数和方法的测试可归为单元测试。单元测试关注代码的实现逻辑，例如，一个 if 语句分支或一个 for 循环的实现。所以，单元测试应尽可能覆盖代码逻辑的每条路径。

2 接口基础

本章讲什么：

（1）什么是接口、接口的分类，以及有哪些常用的 HTTP 接口；

（2）接口测试；

（3）基于 Python+Django+MySQL 的 HTTP 接口实例分析。

2.1 什么是接口

关于什么是接口、什么是 API，如果读者去网上查询，可以找到的解释有很多，而且绝大多数都是很标准的官方解释，例如：

> API（Application Programming Interface）即应用程序接口。你可以认为 API 是一个软件组件，或是一个 Web 服务与外界进行交互的接口。这里的接口可以和 API 画等号。

作为初学者，或者对接口了解很少的读者，这个解释其实并不好理解。那如何才能给读者解释清楚接口呢？如何由表及里地让读者理解接口的运转机制呢？下面笔者结合多年的接口测试经验，用逐层叠加的方式来解释。

1．从功能层面来说

从功能层面来说，可以将接口简单理解为一个黑盒子。其上游负责输入参数，下游负责输出参数，类似于平时的黑盒测试对象，如图 2-1-1 所示。

图 2-1-1　接口理解 1

这里以一个例子来说明：

（1）在 Chrome 浏览器中输入：https://www.v2ex.com/api/nodes/show.json?name=Python，按“Enter”键之后能看到如图 2-1-2 所示的数据。

（2）下面来分析这个过程。

在输入 URL 地址并按 Enter 键后，页面实际发送了一次接口请求。具体的请求是：接口地址（https://www.v2ex.com/api/nodes/show.json?）+请求参数及其值（name=python）。后面这个“name=python”就是输入数据；返回的数据就是浏览器展示的一个 JSON 格式数据。至于这个数据是怎么来的，目前还是看不到的。所以这就像黑盒子一样，输入不

同的数据会得到不同的返回结果。这里读者也可以试试其他的参数，看看返回的数据内容是什么样子的，比如：

https://www.v2ex.com/api/nodes/show.json?name=java

https://www.v2ex.com/api/nodes/show.json?name=测试

图 2-1-2　接口返回数据

名词解释

（1）JSON（JavaScript Object Notation）：是一种轻量级的数据交换格式，独立于语言和平台。JSON 解释器和 JSON 库支持不同的编程语言，是以{}括起来的键值对数据。

JSON 数据格式如下：

```
{"name":"Michael","age":24}
```

在这个例子中，键是字符串形式，值可以取任意类型。name 是键名，Michael 是键对应的值。JSON 是可以嵌套的，比如{"name":"Michael", "birthday": {"month":8,"day":26}}。

（2）JSON 格式的在线检查工具：

https://www.beJSON.com/

https://www.beJSON.com/JSONviewernew/

通过上面的在线检查工具，能检查 JSON 格式的正确性。

2．从数据流层面来说

从数据流层面来说，可以将接口理解为连接前端（Web 页面、APP 等）和数据库（Database）等后端的纽带，用于在二者之间传递数据、处理数据，如图 2-1-3 所示。

图 2-1-3 接口理解 2

在现在主流的框架结构中，后端一般都使用数据库来存储数据，而前端不能直接去数据库中操作数据，一方面不安全，另一方面效率低。要完成数据的交互，必然要有中间的纽带，那就是接口。所以，从这个层面来说，接口主要负责前端页面和后端数据库之间的数据传输和处理。

当下大部分的互联网产品都采用前/后端分离的方式，即前端的表示层负责展示数据及其样式，后端的数据层负责数据处理和存储，如图 2-1-4 所示，它们之间的业务逻辑层负责处理业务逻辑，其中最重要的就是接口。所以说，接口的健壮性、稳定性、抗压性直接决定了这个互联网产品的成败。前端页面通过调用接口传递约定的参数；接口收到请求后，会按照业务逻辑在后端进行处理，并将处理结果和需要的数据返回给前端；前端解析数据并展示在页面。这就完成了广义上的一次前/后端交互。

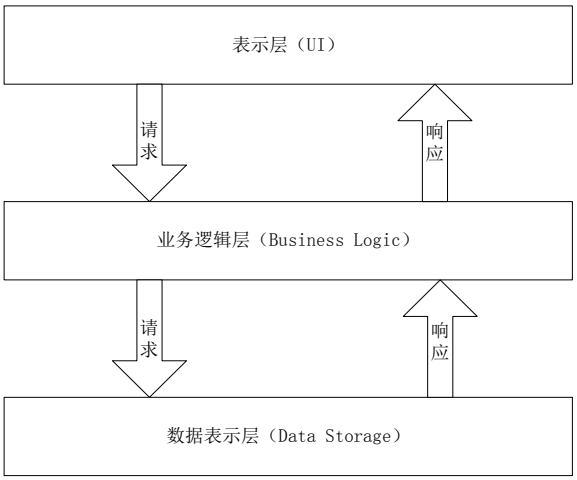


图 2-1-4　三层结构

名词解释：

（1）前端（前台）：可以直观看到和使用的内容都可被归为前端。比如，Web 页面上一些可见的、可操作的界面。

（2）后端：不能被用户可见的内容。其实，后端是一个编程上的概念，具体是指业务逻辑和数据的处理。

（3）后台：通俗意义上是指管理系统，主要用来增加、删除、修改、查询数据，实际上是内部人士使用的一个 Web 系统。

3．从编程层面来说

从编程层面来说，可以将接口理解为业务逻辑处理方法的外在表现形式，如图 2-1-5 所示。它可以是一个类下面的方法，也可以是一个函数。

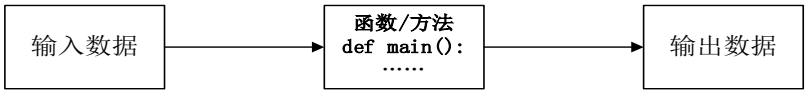


图 2-1-5　接口理解 3

从数据流层面来理解，接口会按照“业务逻辑”处理数据。那么业务逻辑在哪里呢？很显然，是在程序代码的函数或方法中。函数或方法按照逻辑返回不同的数据，这便是接口在不同参数下的不同返回信息。所以，从“白盒操作”角度来看，接口测试是直接对函数或方法的代码层进行测试。

以上从三个层面解释了什么是接口。读者没必要把接口想得过于复杂，认为接口和接口测试多么高深。实际上，接口和日常的页面测试没有本质的区别，只是在外在表现形式和测试方法上略微有差别。读者从以上三个层面可以窥探出接口测试的一般思路，后面还会逐步讲解，请耐心往下看。

2.2 接口的分类

根据系统调用方式可以将接口分为以下两类。

1. 系统之间的接口

系统之间的接口如图 2-2-1 所示。我们用得最多的是第三方接口，比如要做一个系统来展示每天的天气，那天气数据是怎么得到的呢？不可能自己去预测天气，有免费的第三方接口可使用，只需按照接口协议调用想要的天气数据。当然，这是调用系统外部的数据。在系统内部也存在这种调用关系，道理类似。

图 2-2-1 接口分类 1

2. 服务之间的接口

目前主流的系统架构如图 2-2-2 所示，即应用层、服务层和数据层。

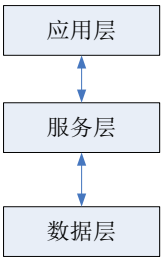


图 2-2-2　接口分类 2

- 应用层：负责展示数据和发起数据请求。比如，12306 购票网站上显示的票数、购买操作等。
- 服务层：为应用层提供数据处理。
- 数据层：用来存储数据，有关系型数据库等。

各层之间的调用过程是怎样的呢？例如，在 12306 网站上买票，首先用户需要选择票，然后通过单击“确定”按钮下单。用户下单就是调用了应用层的接口，假设叫“购买接口”，购买接口会去数据层的数据库中进行查询、新增购买记录等操作。成功完成后，会返回一个成功标志和其他信息。最后，应用层接收到这个接口返回的数据，将买票结果展现给用户。

在这个过程中，各层之间的交互就是通过接口。应用层和服务层之间是通过 HTTP 接口，服务层和数据层主要通过 DAO（Data Access Object）访问数据。在第 5 章讲到用 Python 操作 MySQL 数据库时，使用的 PyMySQL 就是起这个作用的。

2.3　HTTP 接口

下面介绍最常用的 HTTP 接口。

（1）HTTP 接口的应用场景：Web 网站、公司的 OA 服务、小型手机游戏等。

（2）HTTP 请求由三部分组成，分别是：请求地址、消息报头、请求正文。

（3）HTTP 响应也是由三个部分组成，分别是：状态码、消息报头、响应正文。

下面用具体的实例来说明 HTTP 接口。

这里使用的是 Fiddler 工具。该工具可以抓取 HTTP 数据包，辅助进行测试。通过该工具抓取特定网页的协议数据包，来分析 HTTP 的知识点。该工具的具体使用方法会在第 3 章详细讲到，目前读者只需要知道该工具可以抓取 HTTP 数据包即可。

下面抓取的是百度个人中心首页的“百度新闻”→“个性推荐”栏目中的短数据，页面如图 2-3-1 所示。

图 2-3-1 百度个人中心

使用 Fiddler 工具获取的接口数据如图 2-3-2 所示。

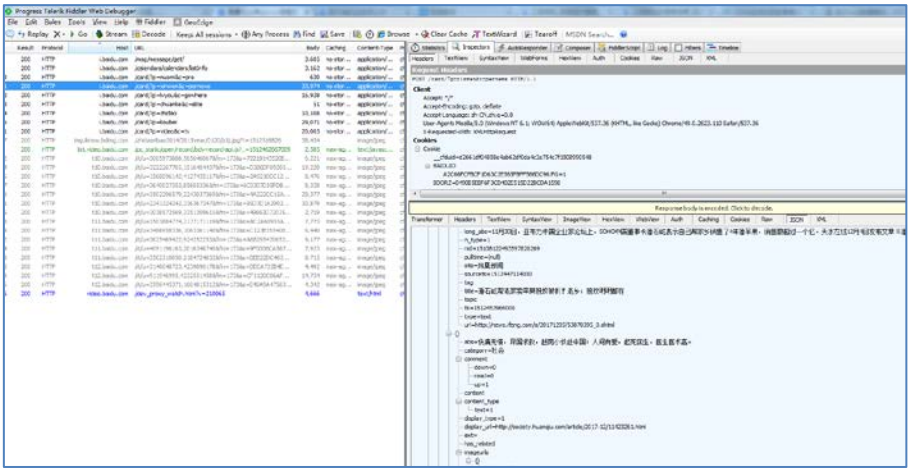


图 2-3-2 使用 Fiddler 工具获取的接口数据

1．HTTP 请求的三个重要数据

（1）请求地址：http://i.baidu.com/card。

（2）消息报头：

```
Host: i.baidu.com
Connection: keep-alive
Content-Length: 0
Accept: */*
Origin: http://i.baidu.com
X-Requested-With: XMLhttp_request
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 （KHTML,
like Gecko） Chrome/49.0.2623.110 Safari/537.36
Referer: http://i.baidu.com/
Accept-Encoding: gzip, deflate
Accept-Language: zh-CN,zh;q=0.8
Cookie:
BDUSS=d0b2JQSGxBbXQ3NUhIcWpqT21xdHVGY3gyTFJENklFbWhSNjRuSmFLOTB4emRaSVFBQ
UFBJCQAAAAAAAAAAAEAAAD0njcitPPJ9E43
```

名词解释

下面介绍消息报头的主要参数。

- Host: i.baidu.com

指定被请求资源的 Internet 主机和端口号，默认是 80 端口。

- Connection: keep-alive

HTTP 长连接（持久连接），客户端和服务器端建立一次连接之后，可以在这条连接上进行多次请求/响应操作。持久连接，可以设置过期时间（Keep-Alive: timeout=60），也可以不设置。

- Accept: */*

表示浏览器能接受任何类型的文件。

- X-Requested-With: XMLHttpRequest

表明这是一次 Ajax 请求（异步），非传统请求（同步）。

- User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64)

用户代理。告知服务器请求方所使用的操作系统及版本、CPU 类型、浏览器及版本、浏览器渲染引擎、浏览器语言、浏览器插件等。

- Accept-Encoding: gzip, deflate

设置从网站中接收的返回数据是否进行 gzip 压缩。

- Cookie: BDUSS=d0b2JQSGxBbXQ3NUhIcWpqT21xd

储存在本地的缓存数据，随 HTTP 请求一起发送，用来给服务器端验证，比如是否已经登录。

（3）请求正文：p=xinwen&c=pernews。

2．HTTP 响应的三个重要数据

（1）状态码：

```
HTTP/1.1 200 OK
```

（2）消息报头：

```
Server: nginx
Date: Tue, 05 Dec 2017 08:31:08 GMT
Content-Type: application/JSON; charset=UTF-8
Transfer-Encoding: chunked
Connection: keep-alive
X-Powered-By: PHP/5.3.24
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check
=0
Pragma: no-cache
```

（3）响应正文：


```
{"status":0,"statusInfo":"","data":{"news":[{"nid":"15108122493597820269
","sourcets":"1512447114000","ts":"1512453966000","title":"\u6f58\u77f3\u
5c79\u5e2e\u8001\u5bb6\u5356\u82f9\u679c\u7956\u575f\u88ab\u6252\uff1f\u8
001\u4e61\uff1a\u7956\u575f\u660e\u660e\u90fd\u5728","url":"http:\/\/news
.ifeng.com\/a\/20171205\/53870395_0.shtml","imageurls":[{"url":"http:\/\/
t10.baidu.com\/it\/u=3005973888,565646067&fm=173&s=72219143520B0753D380B4
B70300D041&w=218&h=146&img.JPEG","height":146,"width":218,"url_webp":"htt
ps:\/\/timg01.bdimg.com\/timg?news&quality=80&size=f218_146&wh_rate=0&img
type=4&sec=0&di=3b739c57cd68d41d10045ba2acefcd10&er=1&src=http%3A%2F%2Ft1
0.baidu.com%2Fit%2Fu%3D3005973888%2C565646067%26fm%3D173%26s%3D7221914352
0B0753D380B4B70300D041%26w%3D218%26h%3D146%26img.JPEG"}...
```


下面介绍响应的消息报头。

“Content-Type: application/JSON; charset=UTF-8”表示服务器发送的实体数据的数据类型。

在 HTTP 请求的返回数据包中，响应状态码分为以下 5 种。

- 1xx：消息。一般是告诉客户端，请求已经收到了，正在处理，别急……
- 2xx：处理成功。一般表示请求收悉、我明白你要的、请求已受理、已经处理完成等信息。
- 3xx：重定向到其他地方。它让客户端再发起一个请求，以完成整个处理过程。
- 4xx：处理发生错误，错误来自客户端。例如，客户端请求的是一个不存在的资源、客户端未被授权、禁止访问等。
- 5xx：处理发生错误，错误来自服务器端。例如，服务器端抛出异常、路由出错、HTTP 版本不支持等。

2.3.1 HTTP 发送请求的方式

HTTP 中有四种发送请求的方式：GET、POST、PUT 和 DELETE。

（1）GET：向特定的资源发出请求。

（2）POST：向指定资源提交“数据进行处理”请求（例如，提交表单或者上传文件），数据被包含在请求体中。POST 请求可能导致新的资源的创建，以及（也可能是“或”）已有资源的修改。

（3）PUT：向指定资源位置上传其最新内容。

（4）DELETE：请求服务器执行删除操作。

在实际应用中常用的是 GET 和 POST。其他的请求方式都可以通过这两种方式间接地实现。

2.3.2 GET 方式和 POST 方式的区别

HTTP 发送请求最主要的两个方式是 GET 和 POST，这两者有哪些区别呢？

区别一：对请求参数的处理方式不同（直观的区别）

（1）GET 请求：请求的数据会附加在 URL 之后，以“?”分隔 URL 和传输数据，如有多个参数则用“&”连接。URL 采用的是 ASCII 编码格式，而不是 Unicode 编码格式，即所有的非 ASCII 字符都要在编码之后再传输。

举例：https://www.v2ex.com/api/nodes/show.JSON?name=Python

（2）POST 请求：POST 请求会把请求的数据放置在 HTTP 请求包的 Body 数据中，数据包的形式可以是“参数名 1=参数值 1&参数名 2=参数值 2”，也可以是 JSON 格式（键值对）。当然，JSON 格式是一种通用的方式。

举例：http://192.168.1.171:8081/api/user_sign/

Body 数据：{"time":"1499933825","sign":"deb697c7fffcca828a7a03a218b2cda5"}

区别二：传输数据的大小不同

HTTP 没有对传输数据的大小进行限制，也没有对 URL 的长度进行限制。而在实际的程序开发中，存在以下限制。

- GET：特定浏览器和服务器对 URL 的长度有限制。例如，IE 对 URL 长度的限制是 2083Byte（2×1024Byte+35Byte）。其他浏览器（如 Netscape、FireFox 等）在理论上没有长度的限制，其限制取决于操作系统的支持。因此，在采用 GET 方式提交数据时，传输数据会受到 URL 长度的限制。
- POST：由于不是通过 URL 传值，在理论上数据的大小不受限制。但实际上，各个 Web 服务器会对采用 POST 方式提交的数据的大小进行限制，例如，Apache、IIS6 都有各自的配置。

区别三：安全性不同

POST 方式的安全性比 GET 方式的安全性高。使用 GET 方式时，在地址栏里可以直接看到请求数据，采用这种方式可能受到 Cross-site request forgery 攻击。

POST 方式需要抓包才能获取到数据，变相地提高了安全性。

2.4 接口测试

2.4.1 什么是接口测试

接口测试主要用于检测外部系统与内部系统之间，以及系统内部各个子系统之间的交互点。其测试的重点是，检查数据的交换、传递和控制管理过程，以及系统间的逻辑依赖关系等。

2.4.2 为什么要做接口测试

1．传统的测试方法成本急剧增加，且测试效率大幅下降

如今的系统复杂度不断上升，传统的测试方法成本急剧增加，且测试效率大幅下降，所以要做接口测试。

另外，接口测试相对容易实现自动化，且接口自动化也比较稳定，可以减少人工测试的人力成本与时间，缩短测试周期，支持后端版本的快速迭代。

2．可以发现很多页面操作中发现不了的 Bug

如果在页面中对输入框做了“必填”限制，则用户不输入内容是不能发送请求和调用接口的，这样通过页面进行测试受到的限制比较多，而直接调用接口则跳过了页面的限制。此时，如果接口没有做限制，则可以绕过前端页面去请求服务器，自然能发现很多页面操作发现不了的 Bug。

3．可以检查系统的异常处理能力

举例说明，在输入框中输入关键字进行搜索，如果前端做了限制，一旦输入的关键字达到一定长度就会被截断了。而在该情况下，调用接口是正常的，且调用接口可以传很长的参数值。此时能发现一些接口层面的 Bug。比如，接口可能会抛出和数据库表有关的日志信息，借此能看到数据库表中的一些字段数据。

4．可以检查系统的安全性、稳定性

举例说明，比如在页面的搜索框中输入特殊的 SQL 注入语句进行搜索时，发现前端会过滤这些 SQL 语句，那么从前端页面的角度来看这是没有问题的。但是，如果接口没有做类似的处理，一旦被他人获取了接口地址并实施 SQL 注入，则会带来严重的后果。所以，页面要做测试，接口更要做测试。

在前/后端分离时，只要前、后端严格按照接口协议来，一般情况下，后端完成接口测试后便可保证业务逻辑的正确性，剩下的便是前端如何展示的问题。所以，一般情况下都是后端先上线，前端再上线。

2.4.3 如何开展接口测试

做接口测试所要面对的现实情况如下：

（1）页面原型还不完整，甚至没有原型设计。

（2）有具体的页面需求文档和要实现的功能说明书。

（3）有接口文档和业务逻辑设计图等。

综上所述，接口测试人员不但需要有很强的抽象能力，而且要有丰富的联想能力。可以按照下面的步骤来实施接口测试（手工）。

（1）获取待测试接口相关数据。一般由开发人员提供接口文档，该文档中包含以下几个基本要素：接口地址、接口请求参数及其说明、请求方式、返回包数据示例、返回码解释等。

（2）充分理解接口逻辑。从产品人员的角度和开发人员的角度，理解接口所要实现的功能、数据的处理逻辑和存储逻辑。该环节尤为重要，需要考量以下几个方面：

- 每个接口所要关联的业务场景是怎样的（从产品的角度）；
- 每个接口的业务处理逻辑和数据存储结构（从开发角度）。

（3）设计接口测试用例。

（4）使用工具模拟发送接口请求，检查返回包数据。

（5）对比预期结果与实际结果，判断接口测试用例的通过性。

2.4.4 前/后端交互的“契约—接口”文档

在实际的分层项目开发中，经常会看到这样的场景：

前端人员 A：“后端人员 B，你什么时候把登录接口给我，我要开始写页面了。”

后端人员 B：“等一下，我改一下接口文档就发给你，你按照这个请求就可以了。”

……

前端人员 A：“后端人员 B，登录用户不存在，你怎么给我返回了 0000 啊，害得我判断错了。”

后端人员 B：“不好意思，接口文档写错了，其实代码里面不是这个返回

码，应该是 1000，稍等，我来改一下文档，你按照最新的文档做判断就可以了。”

在前/后端人员的交互中，能看到反复提到了“接口文档”，那么接口文档是什么呢？为什么它在前/后端之间如此重要？

1．什么是接口文档？

接口文档就是前/后端之间数据交互的一纸契约，有规范的格式和内容要求。后端按照接口协议接收前端传递来的合法数据并返回符合规范的数据，前端按照接口协议传递符合规范的数据并对后端返回的数据依据展示需要做处理。所以说，接口文档是对前/后端开发的强有力约束。

2．为什么接口文档很重要？

（1）接口文档是纽带。当接口文档确定之后，前/后端人员就可以各自开发自己的代码，在开发完成后就可以联调了，而联调的过程就是对接口是否能使用进行测试。这样可以节省前/后端等待的时间。

（2）接口文档是对业务逻辑的传承。在标准的研发流程中，接口文档始终是最新的，所有的前/后端人员修改方案都要先设计接口并更新接口文档，然后再修改代码。这样间接地节省了后期维护的成本和新入职员工的学习成本。

3．接口文档是什么样子的？

接口文档主要包含以下内容。

（1）接口功能：对接口作用的大概描述，使人一眼就知道该接口的作用。

（2）接口 URL：即接口的请求地址。一般都是相对地址，便于在不同环境之间切换。

（3）请求方法：一般的 HTTP 的请求方法是 POST 或 GET。

（4）请求参数：包含参数类型及其限制条件。

（5）返回包数据的实例。

（6）返回码的解释。

图 2-4-1 所示是一个简洁版的接口文档。

接口名称	搜索内容
接口功能	按照内容搜索数据返回
接口URL	http://192.168.10.84:8082/api/search_name
请求方式	GET
请求参数	content　string(8) #搜索内容关键字
返回值	{ "message": "success", "ReturnCode": "0000", "data": [{ "status": 0, "start_time": "2017-07-11T14:22:47", "address": "这是一个test环境", "id": 1, "name": "test" }] }
返回码	0000 查询成功 0001 必填参数为空
说明	

图 2-4-1　接口文档

2.5　接口实例

本节主要分析一个接口实例，目的是让读者直观地理解接口、接口的表现方式、接口的数据处理逻辑。

该实例是基于 Python+Django+MySQL 开发的，目前暂不提供外网访问功能，只能在本地演示，其涵盖了前端、后端、数据库及其交互功能，是一个浓缩版的小系统。

2.5.1　前端页面

图 2-5-1 所示的是一个简单的前端页面，它分为两部分：上方是名称输入框和“搜索”

按钮，可以按照名称来搜索数据；下方是搜索结果展示列表。

图 2-5-1　演示版页面

大家应该比较熟悉这个场景，无论是 Web 页面还是 APP 应用程序，都有类似的搜索功能及其结果展示列表。

用户在这个页面中输入关键字“test”并单击“搜索”按钮，然后在搜索结果展示列表中会显示出搜索到的内容，如图 2-5-2 所示。

图 2-5-2　搜索到的内容

2.5.2 节会以一张流程图来展示这个页面和底层数据的交互过程。

2.5.2　数据流图

图 2-5-3 是按照 2.5.1 小节介绍的业务流程绘制的简单数据流图。

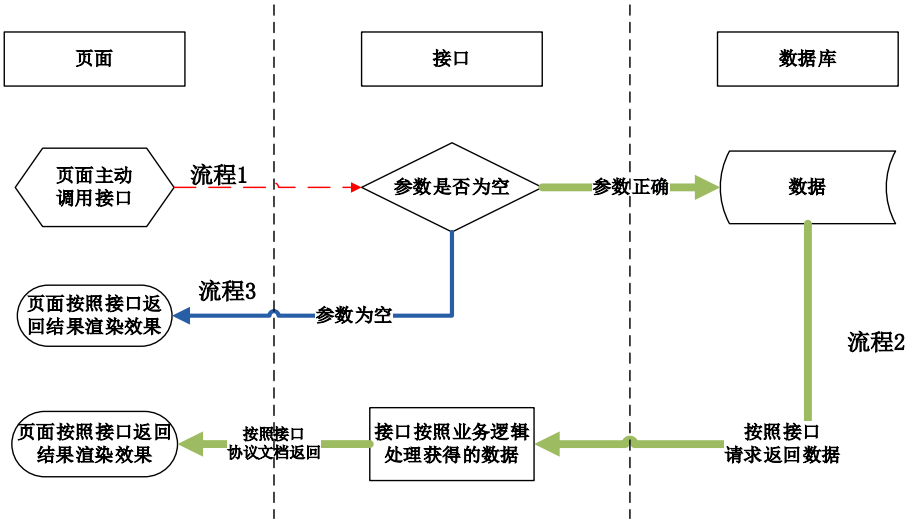


图 2-5-3　数据流图

按照前面对接口的理解，这里将这个数据流划分为页面、接口、数据库 3 部分。下面分别介绍图中的 3 个流程。

1．流程 1

首先，用户在页面主动发起搜索功能，即输入关键字后单击“搜索”按钮，此时页面调用接口，传入接口需要的参数（content）和值（搜索关键字），等待接口返回数据。

这种等待过程很常见，比如在百度中搜一个关键字，然后页面一直在“转圈”，这可能就是页面在等待接口返回数据。出现这种情况，可能是接口返回数据慢导致页面一直在等待，也可能是网络传输延迟等。

2．流程 2

如果接口接收到的请求数据是正确且合法的，则会按照接口中的逻辑到数据库中获取数据，在获取到数据后会对数据进行处理，再将处理结果和返回码按照接口协议中的格式返回给页面。

页面接收到返回数据后，会按照接口协议和实际需要来渲染效果，比如，页面列表

会展示接口返回的数据，即用户看到的检索结果。

3．流程 3

接口接收到请求后，先判断必要的参数是否存在（比如，按照接口文档约定的参数名称应该是 content，然而传来的参数名称是 name，显然这不是需要的）。如果参数不存在，则接口直接返回空数据和对应的返回码。

页面在接收到返回数据后，会按照约定的情况来处理数据。比如，请求接口时传入的参数值为空，则接口依据逻辑返回必填参数为空，页面在接受到该返回信息时会提示用户输入必填值。这些都是依托接口文档的约束。

那实际情况是不是这样的呢？下面在通过页面发送请求时，使用 Fiddler 工具抓取接口数据，来了解实际请求情况。

图 2-5-4 所示的是获取的页面请求接口情况(即用户在输入搜索关键字后单击“搜索”按钮时产生的数据），其中包含页面请求接口的地址、请求的参数及其值、接口返回的数据。

图 2-5-4　Fiddler 工具获取的页面请求接口情况

通过这个页面，能看到以下信息：

（1）接口的地址是 api/search_name。

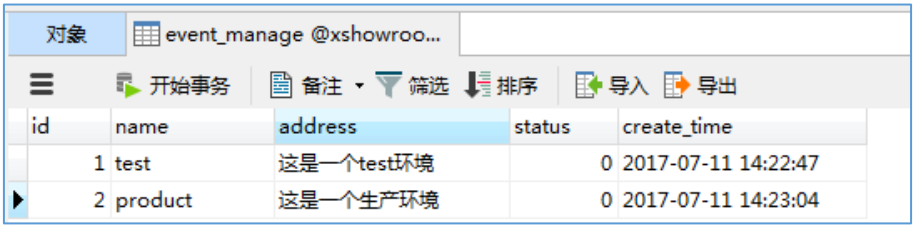
（2）接口请求参数及其值是 content=test。

（3）接口请求方式是 GET。

（4）接口返回数据是 JSON 形式，包括返回码 ReturnCode、返回信息 message，以及返回数据 data。data 中是一个个 JSON 格式的列表数据。这里涉及 Python 的一些语法，会在后面章的语法部分讲到。

大家注意观察，这个 data 中的数据和在页面中看到的搜索结果非常类似，这说明页面的数据确实是这个接口返回的。有时页面在渲染数据时做了取舍，并不是将接口返回的所有数据都展示在页面上，比如这里的 status 参数值没有做展示。

底层的数据结构又是什么样子的呢？请看图 2-5-5。



id	name	address	status	create_time
1	test	这是一个test环境	0	2017-07-11 14:22:47
2	product	这是一个生产环境	0	2017-07-11 14:23:04

图 2-5-5　数据库表结构

图 2-5-5 展示的是底层数据在数据库的存储情况。这里简单地用一张表来存储所有的数据，在实际项目中会有 *N* 张表，每张表之间又会有一些关联字段来建立关系。可以看到，数据库中有两条数据，而接口只返回了其中第一条。原因是，页面搜索的关键字是 test，所以接口的业务逻辑只取了第一条数据。

到这里已经通过接口把前端页面和数据库有机地结合起来了，大家也能比较清楚地理解接口在这两者之间起到的作用。

关于 MySQL 数据库的相关知识，会在第 5 章中讲解。此处读者只需知道有这么一个地方用来存储原始数据即可。

2.5.3 逻辑代码

前几节分别介绍了 Web 页面的样式及功能、页面调用的接口、接口请求和返回数据、数据库数据存储，那是不是就结束了呢？细心的读者可能会发现，现在还不知道接口的代码是怎么处理数据的，即，为什么数据库中有两条数据，而接口只返回了 1 条呢。下面就来介绍这个接口对应的代码是怎么获取数据、处理数据和返回数据的。

这里用的是 Django 框架中的映射关系，将接口请求地址与 views.py 中的处理函数关联起来。其大概的意思是：在调用接口 api/search_name 时，Django 框架会去找 search_name_in 函数。urls.py 文件中的具体配置映射关系如下。

代码：urls.py

```
url(r'^api/search_name/$', views.search_name_in,name='search_name_in')
```

具体的业务处理逻辑在 search_name_in 函数中，见下面的代码：

代码：views.py

```
def search_name_in(request):
      #自定义接口，地址是：http://192.168.1.171:8082/api/search_name?content=test
    search_name=request.GET.get('content','')
    if search_name!='':
   event_tuple=db_test.select_all('SELECT * from event_manage where name like'+"'%"+search_name+"%'")
       event_list=[]
       if len(event_tuple)!=0:
          for info in event_tuple:
event_list.append({'id':info[0],'name':info[1],'address':info[2],
                           'status':info[3],'start_time':info[4]})
        return JsonResponse({'ReturnCode':'0000','message':'success','data':event_list})
    else:
         return JsonResponse({'ReturnCode':'0001','message':'required params is null'})
```

下面介绍一下上方代码的逻辑：

（1）代码第 1 行是定义一个函数，函数名为 search_name_in。后面括号中是请求参数，参数 request 是特别针对 HTTP 请求的，里面存储的是 HTTP 请求的所有数据。很显然，这个函数的功能是对 HTTP 请求包中的数据做处理。

（2）代码第 3 行是获取接口请求参数 content 的值，并将其值赋给了 search_name 参数。当这个参数值不存在时，则默认是空字符串。

（3）代码第 5 行是按照业务逻辑获取所有的数据。这里使用 SQL 语句查询数据，并将结果赋给了 event_tuple 参数。

（4）代码第 8、9 行是对获取到的数据进行 for 循环，并将数据组装成一个字典加入 event_list 中，然后返回一个 JSON 数据包。

前面简单地说明了这个接口的处理逻辑。当然，其中有很多 Python 语法和 Django 框架知识，可能有些知识读者不是特别清楚。这些暂时都不重要，只要了解大概的逻辑即可，在后文的代码开发过程中会讲解 Python 语法，所以读者不用担心。

2.6 补充知识点

2.6.1 名词解释

1．Django 框架

Django 是一个开放源代码的 Web 应用框架，由 Python 写成。它采用了 MVC 的框架模式，即模型（M）、视图（V）和控制器（C）。相比其他 Web 框架，Django 的优势是：大而全，集成了 ORM、模型绑定、模板引擎、缓存和 Session 等诸多功能。

2．HTTP

HTTP 即超文本传输协议（Hypertext Transfer Protocol），是基于请求/响应范式的（相当于客户机/服务器）。一台客户机与服务器建立连接后，发送一个请求给服务器；服务器接到请求后，给予相应的响应信息。HTTP 的默认端口是 80，可以不写。

3．MySQL 数据库

MySQL 是一种关系型数据库。它将数据保存在不同的表中，而不是将所有数据放在一个大仓库内，这样增加了运行速度并提高了灵活性。MySQL 有以下特点：

（1）是开源的，无须支付费用就可以直接用。

（2）使用标准的 SQL 数据语言形式。

（3）相对于 Oracle 和 SQL Server，MySQL 更小，更轻量级，当然更适合测试。

2.6.2 答疑

（1）前端页面已经做了“非必填”判断，为什么接口还要做非必填参数的校验？是不是多此一举？

这并不是多此一举，而是双重保护。通常，对于必填参数的校验，前/后端都要做。前端做校验，一方面是给用户友好的提示；另一方面是最直接的系统保护，减少了对后端的请求。而后端做校验，一方面，如果前端没有做保护，则后端不至于出错；另一方面（也是最重要的），如果用户绕过前端的请求直接调用接口则不至于出错。

（2）前端开发、后端开发是什么意思？

- 前端开发。

前端开发一般指的是 Web 前端开发，即网站前端页面（即网页的页面）的开发。简单地说，网站前端工程师负责网站中用户可见的内容开发，如网页上的特效、网页的布局、图片和视频等。网站前端工程师的工作内容是，将美工设计的效果图设计成浏览器可以运行的网页，并和后端开发工程师配合，做网页的数据显示和交互。

- 后端开发。

后端开发一般负责网站后台逻辑的设计和实现，以及用户及网站的数据的保存和读取。比如，在前端实现了登录页面，那么当用户输入账号和密码并单击“登录”按钮时，其实前端已经完成了自己的事件，然后就是等待后端返回账号和密码校验结果，前端根

据这个校验结果来显示登录成功、账号或密码错误等提示信息。

（3）前/后端开发的顺序是什么？

在实际的项目开发中，前/后端开发是并行开展的，它们之间能并行的关键是接口文档，前/后端开发都要依据接口文档来做各自对应的事情，如图 2-6-1 所示。

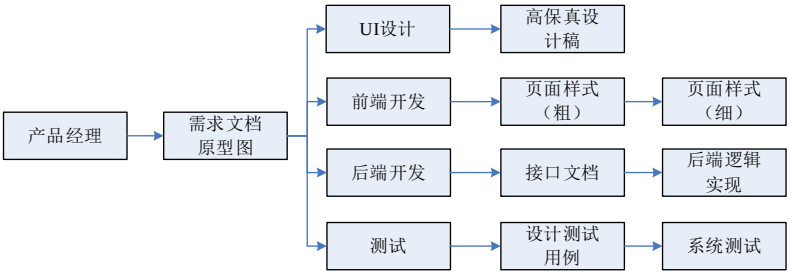


图 2-6-1　前/后端并行开发机制

3 接口手工测试

本章讲什么：

- HTTP 接口工具；
- Fiddler 工具的使用；
- 接口手工测试的用例设计。

第 2 章中，在分析前/后端交互数据时，使用了 Fiddler 工具来捕获 HTTP 接口数据，本章的 3.1 节主要讲解 HTTP 接口工具，重点是 Fiddler 工具，包括以下四个部分：

- 用 Fiddler 工具手工调用 HTTP 接口。
- 用 Fiddler 工具获取 PC 端的网络包数据。
- 用 Fiddler 工具获取手机端的网络包数据。
- 用 Fiddler 工具篡改数据。

对功能测试来说，好的用例设计会起到事半功倍的作用。这个道理对于接口测试同样适用，而且接口测试用例设计还有一套比较成熟的规范可遵循。3.3 节主要讲解接口的手工测试用例设计。

3.1 HTTP 接口工具

目前，HTTP 接口工具主要分为以下几类。

（1）接口手工测试工具：这类工具主要是用来模拟发送 HTTP 请求，并接收接口返回的数据。这类工具包括 Fiddler、Postman、Wireshak、在线 HTTP 地址等，适用于日常的手工测试。

（2）接口自动化测试工具：相比接口的普通测试工具，此类工具能批量处理接口请求，支持断言判断并能生成简单的测试报告，这类工具包括 JMeter、soapUI 等。

（3）接口性能测试工具：主要验证接口性能指标。这类工具包括 LoadRunner、JMeter 和 soapUI 等。

下面简单介绍各种工具的使用方法。

（1）Fiddler：一个 HTTP 抓包工具。Web 测试和手机端测试都可以使用该工具，如图 3-1-1 所示。该工具也能发送 HTTP 请求用来模拟接口测试，详细内容会在后文介绍。

（2）Postman：这是 Chrome 浏览器的一个插件，支持不同接口测试请求，能够管理测试数据和自动化运行，如图 3-1-2 所示。

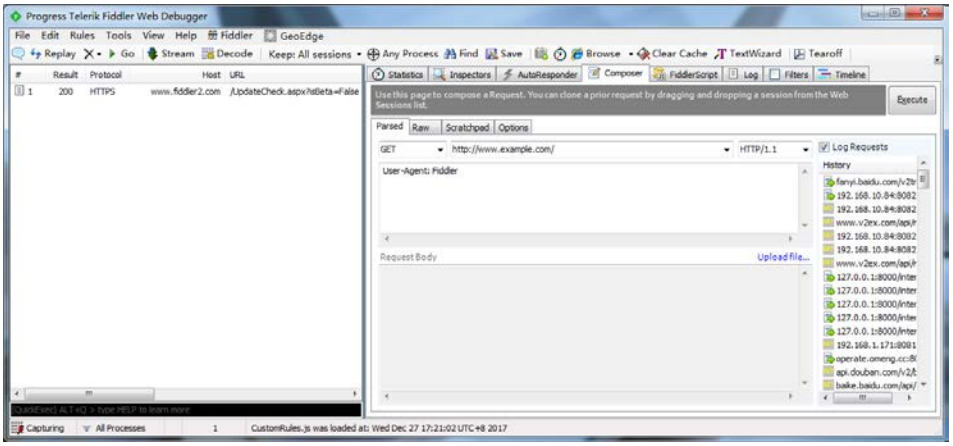


图 3-1-1　Fiddler 界面

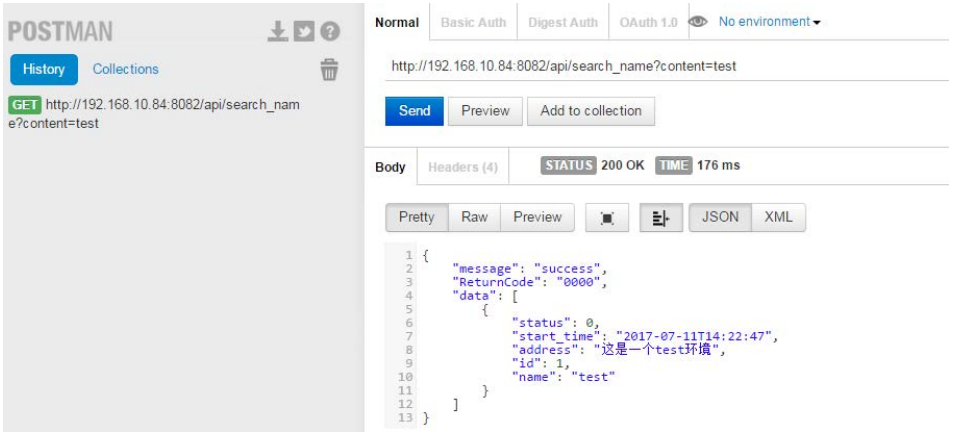


图 3-1-2　Postman 界面

（3）Wireshak：与 Fiddler 的作用相同，都是抓包工具。但是 Wireshak 支持各种协议，包括 TCP、UDP、HTTP 等，比较适合用于进行底层网络数据测试。但如果用来测试接口，则因为数据量太多、刷新太快，不利于准确定位具体操作对应的接口。

（4）在线 HTTP 地址：一些在线的网站也能模拟 HTTP 请求，但是不支持内网环境。

（5）JMeter：以接口性能测试出名。它是用 Java 开发的，易上手，如图 3-1-3 所示。基本上配置 JMeter 后就可以做接口测试。

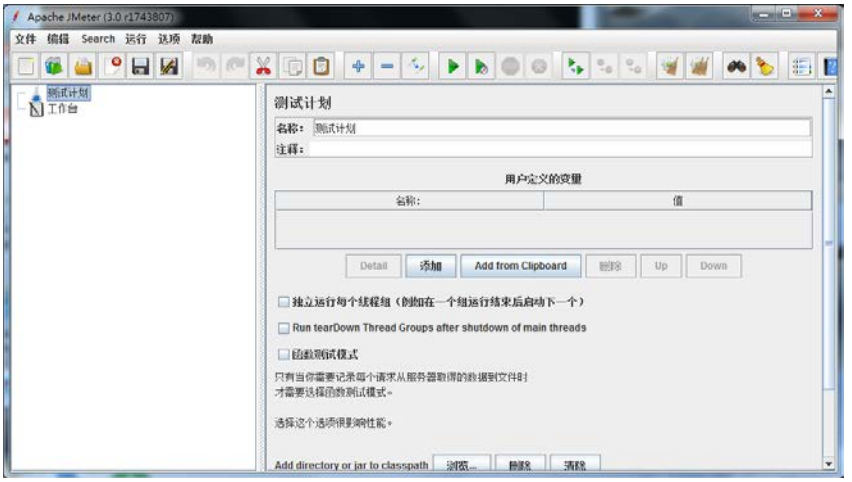


图 3-1-3　JMeter 页面

（6）LoadRunner：一款性能测试利器。代码开发和报告分析是其优势。它属于非开源工具，但是同样可以做接口自动化测试。

（7）soapUI：开源测试工具，通过 soap/http 来检查、调用、实现 Web Service 的功能/负载/符合性测试

（8）自己编写的代码：比如使用 Python 语言和其 requests 包。本书正是使用的这种工具。

既然有那么多好用的 HTTP 接口测试工具，为什么笔者会选择 Fiddler 呢？其主要原因在于 Fiddler 工具的强大性：

- 它是一个本地化的工具，不需要依托其他组件。当然，目前它只支持 Windows 平台。
- 它对 HTTP 支持较好，且操作简单易上手。

提示：

Fiddler 工具具有抓包和分析功能，省去了安装其他工具的麻烦。这一点对于测试人员尤为重要。因为，通过抓包准确地判断问题出在哪一端（前端还是后端），一方面利于开发人员迅速地定位问题，另一方面也是测试人员能力的体现。

- 它功能强大，有截包、篡改数据、限速等功能，能很好地辅助测试。

3.2　Fiddler 工具的使用

3.2.1　Fiddler 工具介绍

1．下载与安装

直接在百度中搜索“Fiddler”，下载安装文件后直接安装。当然，也可以使用免安装版本。

2．Fiddler 工具界面介绍

图 3-2-1 所示的是 Fiddler 工具的主界面。其中显示的是访问百度个人中心时捕获的数据，其结构大致分为 3 个区域。

- 区域 1：标准的 Windows 程序菜单栏。
- 区域 2：捕获的页面请求数据，以列表形式显示出请求数据的大概信息。
- 区域 3：每个请求的详细信息。单击（区域 2）列表可展示相关数据。其分为上部和下部，上部是请求信息，下部是响应信息。

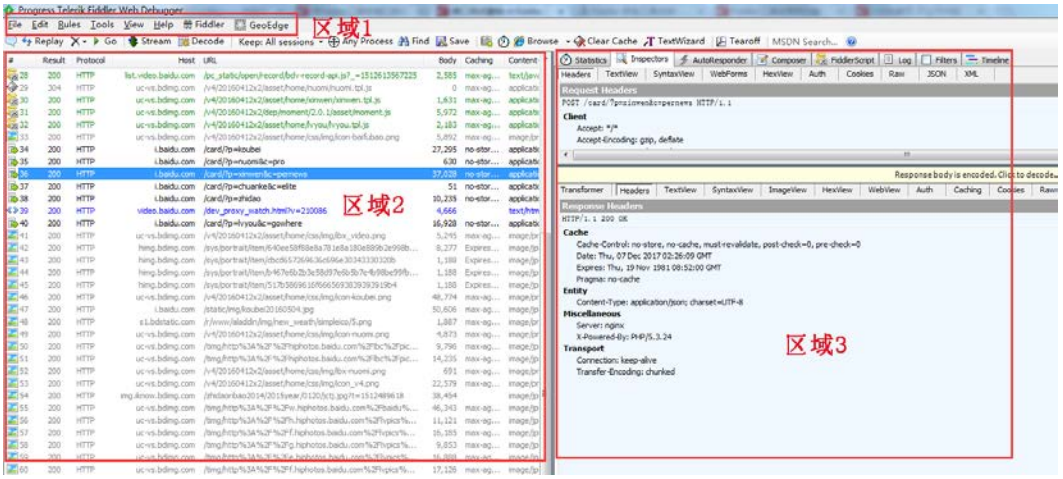


图 3-2-1　Fiddler 工具页面

3．Fiddler 工具的工作原理

Fiddler 工具是一个中转站，也是代理服务，所以它才能获取请求和响应数据，如图 3-2-2 所示。

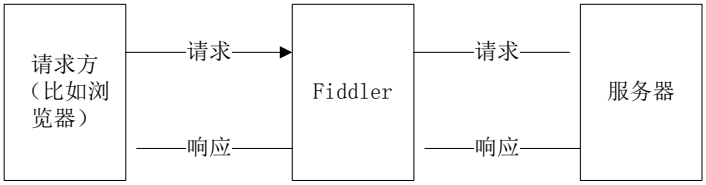


图 3-2-2　Fiddler 工具的工作原理

3.2.2　手工调用 HTTP 接口

使用 Fiddler 工具可以进行单次的 HTTP 接口请求/响应数据分析。比如，有一个接口数据需要做测试（备注：下面的接口是作者本机环境，暂不支持对外访问）。

- 接口地址：http://192.168.10.84:8082/api/search_name。
- 请求参数：content。
- 请求方式：GET。

1．发送 HTTP 请求

操作过程如图 3-2-3 所示。

（1）切换至工具的“Composer”选项卡（打开工具时默认打开该选项）。

（2）选择请求方式。通过下拉列表中获取 GET 方式。

（3）输入接口地址。因为是 GET 方式的，所以地址实际上是“URL+参数”的组合。

（4）输入请求的消息报头。

（5）单击“Execute”按钮，发送一次 HTTP 请求。

（6）在图 3-2-3 左侧列表中检查返回包的概要信息。在本例中，HTTP 返回码是 200（代表正常），HOST 是主机地址，URL 是接口地址段。

（7）在最右侧也有一个列表，标题是 Log Requests，记录的是历史请求接口。双击列表中的数据，可以查看历史请求及其结果。

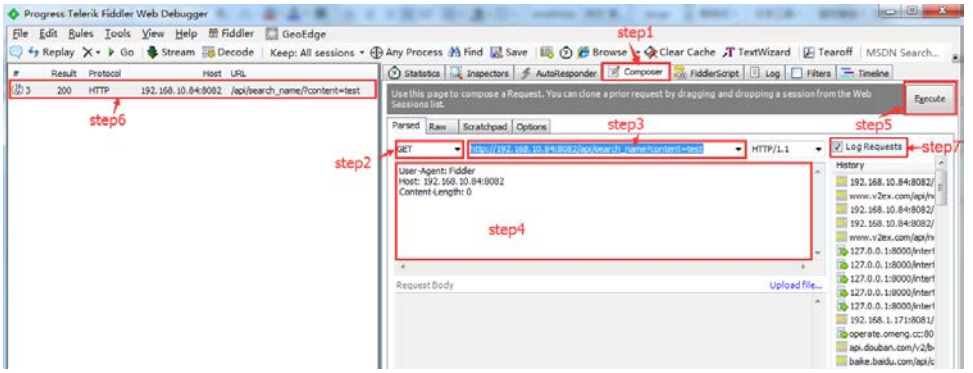


图 3-2-3　用 Fiddler 工具发送 HTTP 请求

接下来查看这次请求的返回数据包。

2．查看返回数据包

操作过程如图 3-2-4 所示。

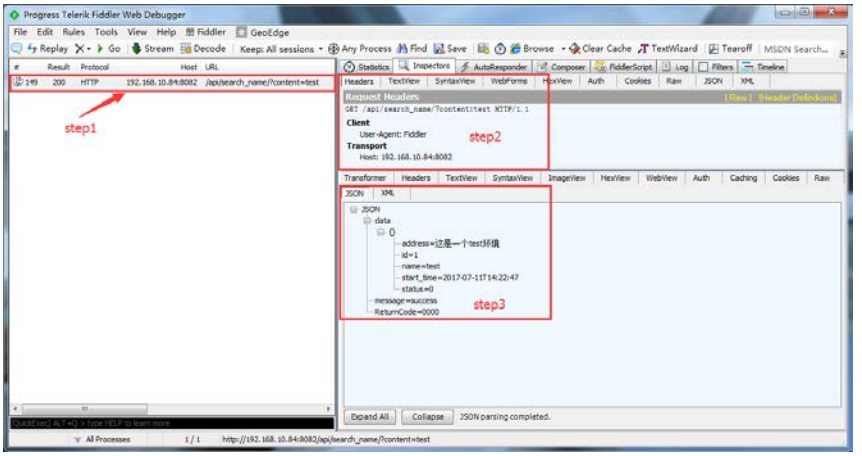


图 3-2-4　在 Fiddler 工具中查看返回数据包

（1）双击图 3-2-3 中左侧的请求列表数据。

（2）在右侧选择“Inspectors”→“Headers”命令。

（3）在下方看到本次接口请求方式、地址段及其参数。

选择 JSON 的意思是“使用 JSON 格式展示返回数据包数据”，能看到并不是所有的返回包数据都是 JSON 格式的，对于非 JSON 格式的数据可以使用 TextView 模式来查看。

下面还有一个接口需要做测试：

- 接口地址：http://192.168.10.84:8082/api/search_name。
- 请求参数：content=test。
- 请求方式：POST。

以下是这个测试的执行步骤，如图 3-2-5 所示。

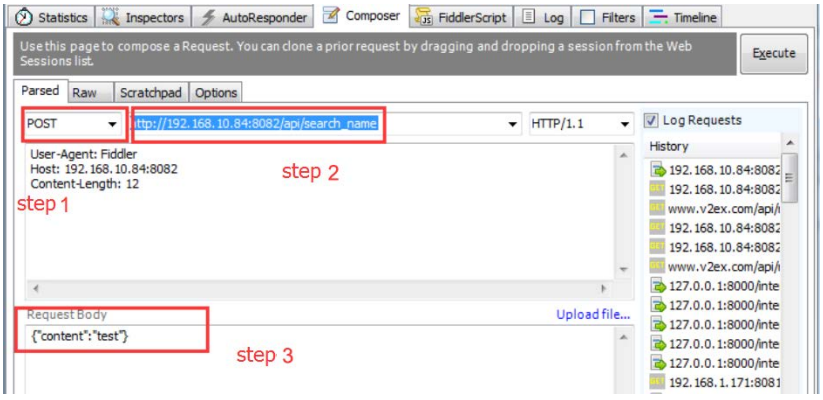


图 3-2-5　用 Fiddler 工具发送 POST 请求

（1）选择执行方式为 POST。

（2）输入纯粹的接口地址，不带请求参数。

（3）在“Request Body”中设置参数。

（4）检查返回包数据的方法与 GET 方式中的“查看数据包”方法是否一样。

3.2.3　获取 PC 端的网络数据包

该方法主要用来获取 PC 端设备的 HTTP 数据包。例如浏览器访问网站时，页面所调用的接口及其返回的数据包。

比如，打开一个陌生的网站，很想知道这个网站都和服务器做了哪些数据交互、都发送了哪些请求，甚至页面执行了某个操作后数据是怎么来的。

这里以第三方购物平台登录页面（http://www.xshowroom.cn/index/login）为例来介绍具体步骤。

（1）打开 Fiddler 工具，确保工具左下角的“Capturing”处于开启状态。

（2）访问网站首页，输入登录账号、密码和验证码，单击“确定”按钮。

（3）观察 Fiddler 工具左侧区域，找到 Host 标题下对应的域名值（建议按 Host 排序来筛选），如图 3-2-6 所示。

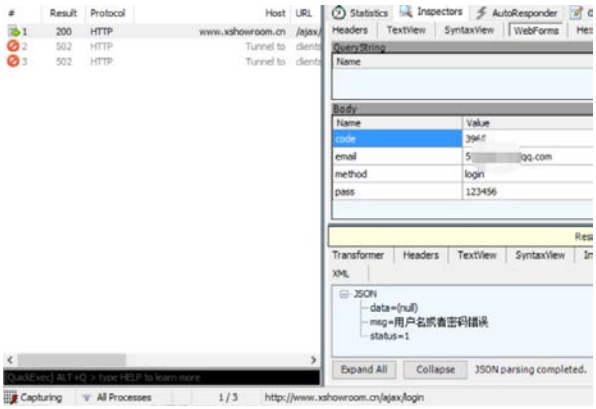


图 3-2-6　Fiddler 捕获的 PC 端数据包

（4）双击左侧的该条数据包概要，右侧展示的是具体请求数据和返回数据。

提示：

（1）左下角的“Capturing”标志：显示则表示开始监听和抓取数据，不显示则表示不监听和抓取（也就不显示数据包）。

（2）“All Processes”标志：监听 PC 端的所有网络请求。如果显示为 Web Browsers（Web 浏览器），则只监听 Web 浏览器发起的请求和返回数据；如果显示为 Non-Browser（非 Web 浏览器），则监听除 Web 浏览器外的所有其他进程；如显示为 Hide All（屏蔽所有），则不监听。

以上两个标志配合使用，可以很方便地控制监控时间和要监听的对象，避免其他无用数据的干扰。

3.2.4 获取手机端的网络数据包

既然 Fiddler 工具可以充当一个代理服务器的角色，自然可以作为手机上网的代理服务器。手机端发送的网络请求，会被代理服务器（Fiddler 工具）所捕获。这也正是 Fiddler 工具能捕获到 APP 数据请求的原因。

配置代理的方法如下：

（1）保证手机连接的 Wi-Fi 和 Fiddler 工具所在电脑处于同一个网段。

（2）将 Fiddler 工具设置为支持移动端连接：选择菜单中的“Fiddler”→“Tools”→“Fiddler Options”命令。在打开的对话框中的“Connections”选项卡里勾选“Allow remote computers to connect”复选框，设置“Fiddler listenson port”为“8888”，如图 3-2-7 所示。单击“OK”按钮关闭 Fiddler 工具，之后重新打开 Fiddler 工具。

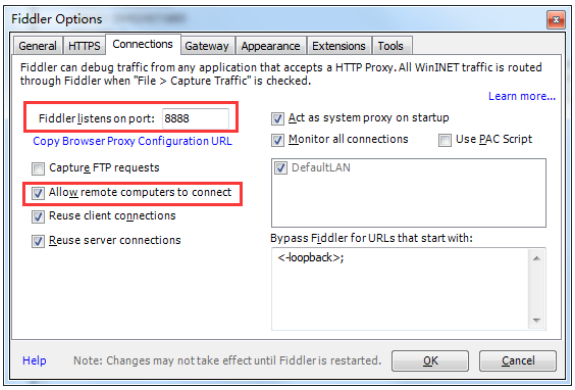


图 3-2-7 Fiddler 工具设置

（3）在“运行”对话框中输入“cmd”，在弹出的 DOS 窗口中执行“netstat -anop tcp”命令，以查看 Fiddler 工具进程是否能正常监听 8888 端口，如图 3-2-8 所示。如果服务没有正常开启，则可以尝试使用其他端口。

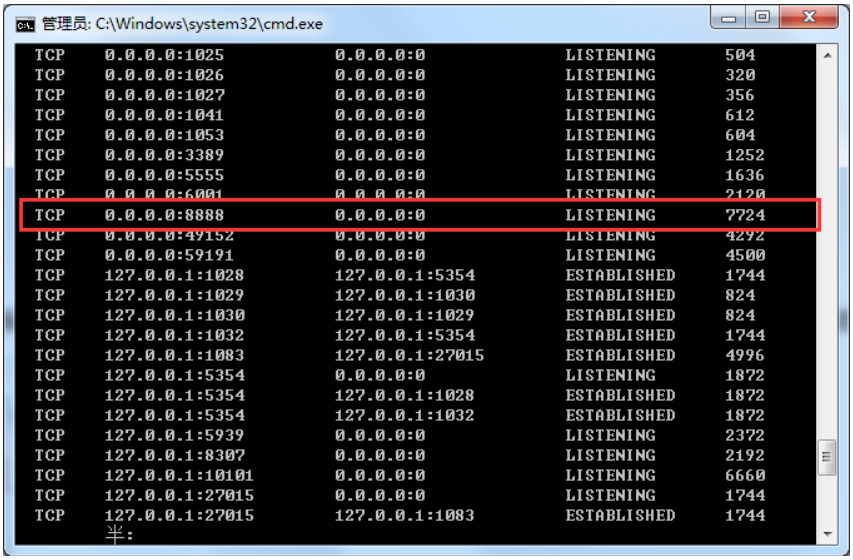


图 3-2-8　Fiddler 工具端口检测

（4）在 DOS 命令窗口中里执行“ipconfig”命令，以查看本机 IP，如图 3-2-9 所示。

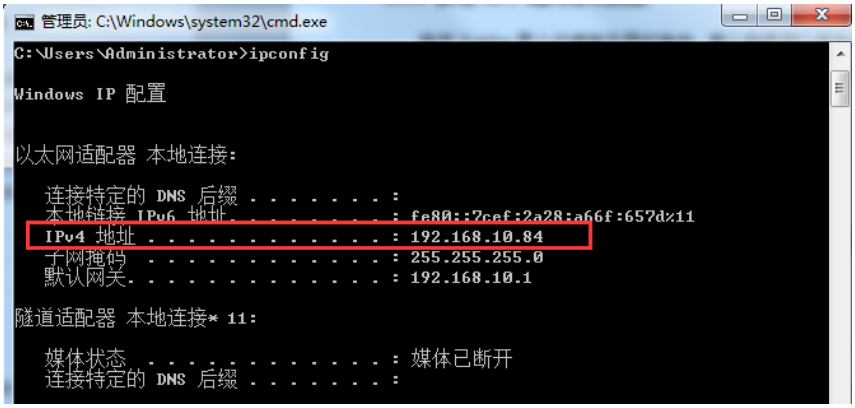


图 3-2-9　查看本机的 IP

（5）设置手机网络代理：

- 打开手机连接的无线，将代理设置为手动。
- 将主机设为 192.168.10.84（运行 Fiddler 工具的电脑 IP）。
- 将端口设为 8888（与 Fiddler 工具中设置的端口保持一致）。
- 单击“确定”按钮（苹果手机直接后退即可）。

（6）验证。在手机上访问 APP，或通过浏览器访问“百度”首页，检查 Fiddler 工具左侧列表中是否有数据。

注意：

（1）测试完成后请记得关闭手机端代理设置。

（2）在做手机端抓包分析时，可以让 Fiddler 工具左下角的“Capturing”不显示，这样 Fiddle 只显示主动发起的请求，而不会截取手机端所有的请求包，可以减少不必要的数据。

3.2.5 截包与改包

在进行具体操作之前，要明确一件事情——为什么要做截包和改包，这对测试有什么帮助?

（1）所谓“截包”，有两个层面的含义。

- 截断请求数据包：即 Fiddler 工具截获页面的请求数据包，使其发送不到服务器端，只到 Fiddler 工具这一层就结束。
- 截断返回数据包：即 Fiddler 工具拿到服务器端的返回数据包，但不返回给前端页面（即返回数据包被拦截），则请求方处于等待服务器端返回数据包的状态。这个可以用来模拟“等待服务器端返回数据包”和“服务器端返回数据包超时”情况，以测试前端页面如何处理。

（2）“改包”的前提是“截包”，分为两种情况：① 修改请求数据包，再提交到服务器端，检查服务器端的处理情况；② 修改返回数据包，再提交给请求方，检查请求方

的处理情况。它主要用在模拟“请求数据包或返回数据包不好被制造”的场景，比如，前端获取到的返回数据包中数据字段值超长时，页面是否会做截断处理（也可以修改数据库数据）。

截包与改包操作如下。

1．场景一：截断请求数据，然后篡改请求数据

（1）在 Fiddler 工具工作界面的左下角单击，出现如图 3-3-10 所示红色的“T”标志，表示工具处于截断请求数据状态。

图 3-2-10　截包设置

（2）访问页面，在如图 3-2-11 所示界面中输入示例参数，然后单击“分析”按钮。

图 3-2-11　输入参数

（3）在 Fiddler 工具左侧列表中双击页面请求的接口，在右侧查看请求数据，如图 3-2-12 所示。将请求数据参数名由“phone”改为“telephone”，然后依次单击 Fiddler 工具工作界面右侧的“Break on Response”和“Run to Completion”，

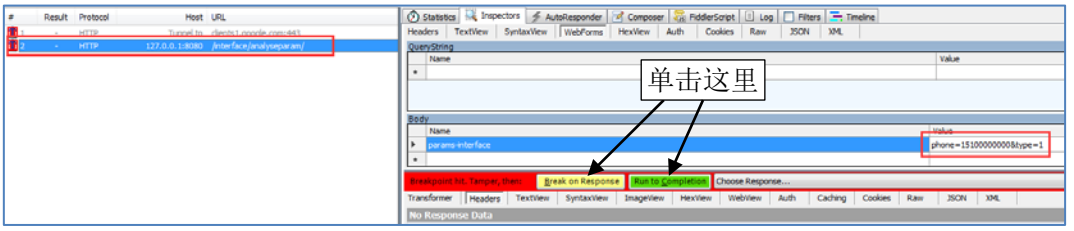


图 3-2-12　将请求数据参数名由“phone”改为“telephone”

（4）结果如图 3-2-13 所示。

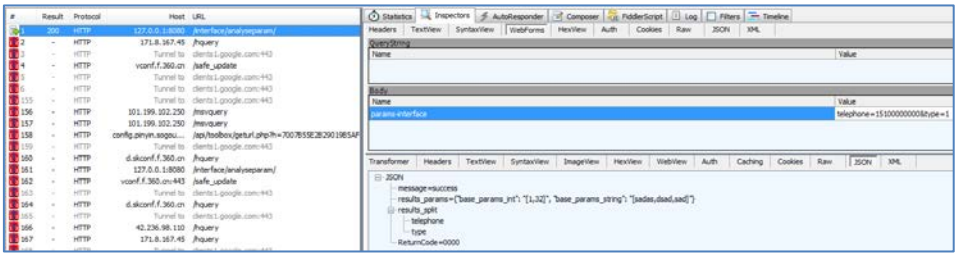


图 3-2-13　修改后的结果

（5）检查接口数据和页面显示的数据，如图 3-2-14 所示。

图 3-2-14　返回结果

总结：通过接口返回的数据和页面能出，已成功将请求参数名称由“phone”修改成了“telephone”。

2．场景二：篡改返回的数据包

（1）在 Fiddler 工具工作界面的左下角双击，出现如图 3-2-15 所示红色的 II 标志，表示工具处于截断返回数据包状态。

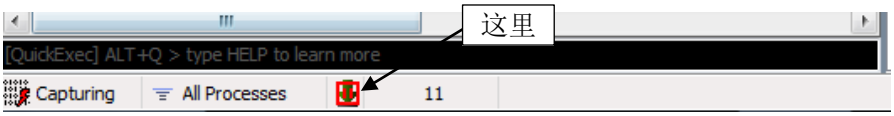


图 3-2-15　处于截断返回数据包状态

（2）访问页面，在如图 3-2-16 所示界面中输入示例参数，单击“分析”按钮。

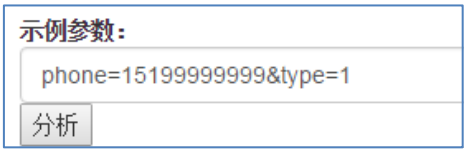


图 3-2-16　输入示例参数

（3）在 Fiddler 工具左侧列表中双击页面请求的接口，在右侧查看接口返回的数据。此时看到 Fiddler 工具已得到了接口返回的数据，但是前端的页面还没有显示，如图 3-2-17 所示，则说明 Fiddler 工具成功地将接口返回数据拦截在工具中了。

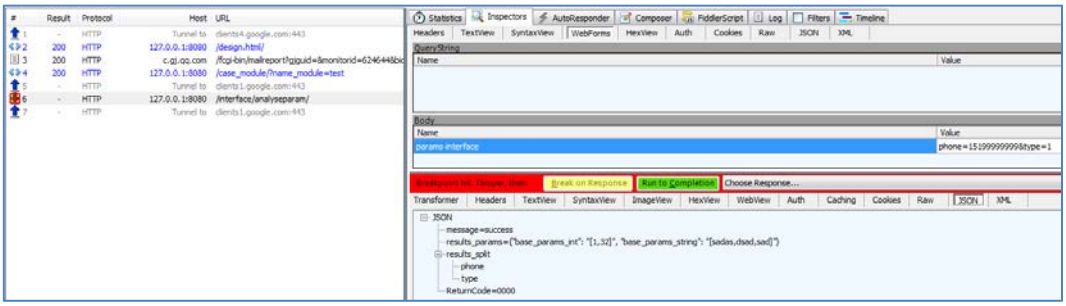


图 3-2-17　截断返回的数据

（4）在 TextView 模式下修改返回的数据（例如修改“type”为“type9999”，如图 3-2-18 所示），然后单击 Fiddler 工具右侧的“Run to Completion”。

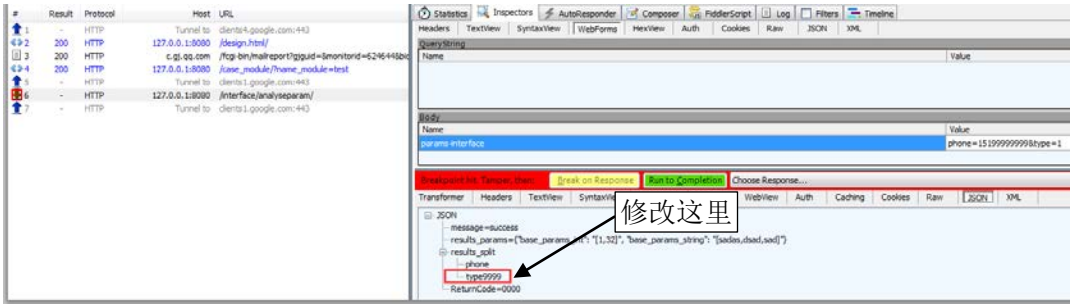


图 3-2-18　修改返回的数据

（5）检查接口数据和页面显示的数据。通过接口返回的数据和页面能看出，已成功将返回参数由“type”改为“type9999”，并在页面中正确显示，如图 3-2-19 所示。

图 3-2-19　返回的结果

3.2.6　Fiddler 工具的其他功能

1．辅助 LoadRunner 录制接口请求

在使用 LoadRunner 进行接口性能测试时，如果不知道如何编写脚本，则可以通过 LoadRunner 录制 Fiddler 工具的方法来获取接口请求数据（此处需要注意，Fiddler 工具必须是 32 位的，因为 LoadRunner 暂时只支持对 32 位应用程序的录制）。

2．模拟低速网络环境

启用方法是：选择菜单中的“Rules”→“Performance”→“Simulate Modem Speeds”命令，如图 3-2-20 所示。

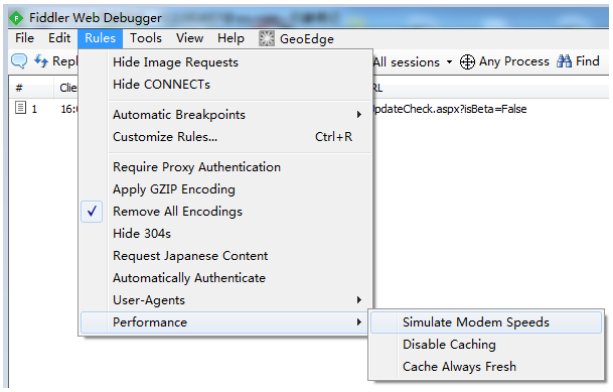


图 3-2-20　低速设置

3.3 接口手工测试的用例设计

测试用例决定了测试效果的好坏。在接口测试中，最重要的部分就是“入参”，即请求参数的组合情况，因为一个接口的返回结果取决于其请求参数。

本书关于接口手工测试用例设计就是围绕“入参”来开展的。

3.3.1 接口测试用例设计——总纲

应该怎样设计接口测试用例呢？可以按照下面步骤来设计。

（1）接口请求参数组合形式，决定了接口的不同返回结果。而接口协议对于接口的每个请求参数及其值都有要求，故这就是第一个维度——参数校验。从参数组合的角度，应尽可能多地覆盖参数组合场景。

（2）接口的功能是业务逻辑的体现，那第二个维度就是“逻辑校验”。应结合“该接口处理的业务逻辑与底层数据存储”来设计对应的场景。这其实也是参数的组合，只不过，这种参数组合是为了校验逻辑而有针对性地设计的。

归纳一下，接口测试用例的设计包含两个维度：参数校验、逻辑校验。

图 3-3-1 所示的是搜索接口的测试版接口文档。下面利用该文档来讲解如何按照两个维度来设计接口测试用例。

提示：接口文档是设计测试用例的重要文档，必须要理解透彻。

3.3.2 接口测试用例设计——参数校验

1. 步骤一：梳理逻辑

对照上面的接口文档和代码逻辑做一轮拆分，会得到下面的关键数据。

（1）接口完整的请求地址是：http://192.168.10.84:8082/api/search_name?content=test。

（2）参数要求：string 类型参数，长度限制为 8 位，必填参数。

接口名称	搜索内容
接口功能	按照内容搜索数据返回
接口URL	http://192.168.10.84:8082/api/search_name
请求方式	GET
请求参数	content string(8) #搜索内容关键字
返回值	{ "message": "success", "ReturnCode": "0000", "data": [{ "status": 0, "start_time": "2017-07-11T14:22:47", "address": "这是一个test环境", "id": 1, "name": "test" }] }
返回码	0000 查询成功 0001 必填参数为空
说明	

图 3-3-1　接口文档

（3）对业务逻辑分析可知，查询内容的语句是：

```
'SELECT * from event_manage where name  like'+"'%"+search_name+"%'"
```

2. 步骤二：分解接口要点

通过对接口入参要求和业务逻辑的了解，可以按照以下几点设计入参。

（1）参数本身：分为必填参数存在、必填参数不存在。

（2）参数值的类型：分为 string 类型、非 string 类型。

（3）参数值的长度：分为长度 7 位、长度 8 位、长度 9 位。

（4）参数值 SQL 注入。

3. 步骤三：设计用例框架

使用 MindManager 或者 Xmind 设计用例框架，如图 3-3-2 所示。

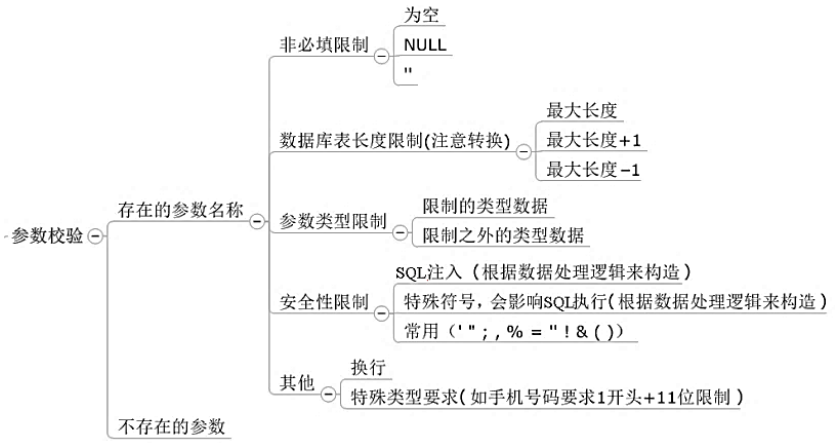


图 3-3-2　参数校验框架

4. 步骤四：生成具体用例数据

在了解了接口入参要求和业务逻辑之后，便可以设计具体的接口测试用例，见表 3-3-1。

表 3-3-1　接口测试用例数据

参数描述	参数举例
必填参数存在（string 类型）	content=test
	content=中文
必填参数存在（int 类型）	content=1
必填参数不存在	content=
	content=''
	content=NULL
长度 7 位	content=sadwdqw
长度 8 位	content=qwewqwqw
SQL 注入	content='or'1=1'or'
其他	content='
	content=!
	content=%

表 3-3-1 中描述的是一个参数时的用例。如果是两个参数，那么应该如何组合场景呢？设计多参数用例时，应尽量保持一个参数在变化，其他参数都不变。比如接口有两个参数，参数 1 的取值有 6 种情况，参数 2 的取值有 5 种情况，那该接口的参数应有 11 种组合情况。

3.3.3 参数校验——SQL 注入

下面重点介绍 SQL 注入，也借机强调一下接口测试的重要性。

（1）表 3-3-1 中接口底层查询内容的语句是：

```
'SELECT * from event_manage where name like'+"'%"+search_name+"%'"
```

（2）如果接口入参是“content=test”，则最终数据库中的执行语句变成：

```
SELECT * from event_manage where name like'%test%'
```

数据库执行该语句，能得到 name 字段中含有 test 的所有数据，这是正确的处理逻辑。

（3）如果接口入参是“content='or'1=1'or'”，且接口没有对特殊字符做处理，则最终数据库中的执行语句变成了：

```
SELECT * from event_manage where name like '%' or '1=1' or '%'
```

数据库执行该语句时，由于“1=1”是 True，则条件判断成立，那这行 SQL 语句与 `SELECT * from event_manage` 的作用是相同的，返回的是 event_manage 表中所有的数据，显然这并不是业务逻辑设计的本意。

（4）实际的接口调用结果是否如分析的一样呢？如图 3-3-3 所示，在调用 pi/search_name 接口时，传递的是参数 content 及其值“'or'1=1'or'”，接口很好地将该参数直接放到 SQL 中去执行了，得到了该数据库中的所有数据。

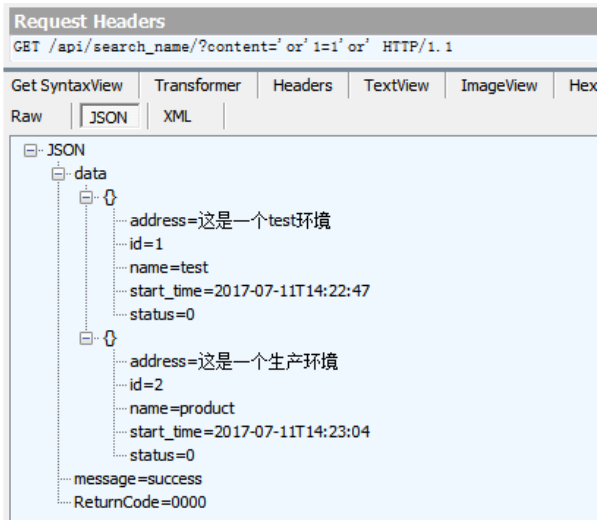
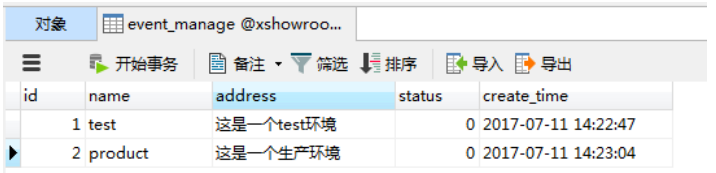


图 3-3-3　接口返回数据包

结论：通过执行含有 SQL 语句的接口请求，该接口成功返回了数据库表中所有的数据，成功实施了一次简单的 SQL 注入。event_manage 表中的数据如图 3-3-4 所示。



id	name	address	status	create_time
1	test	这是一个test环境	0	2017-07-11 14:22:47
2	product	这是一个生产环境	0	2017-07-11 14:23:04

图 3-3-4　event_manage 表中的数据

3.3.4　接口测试用例设计——逻辑校验

由于涉及逻辑，所以这一环节和具体的业务关联性较高。本书并没有列举出通用的业务逻辑检验方法，主要是想强调逻辑校验的重要性。一般可以参照图 3-3-5 所示的思路来设计用例。

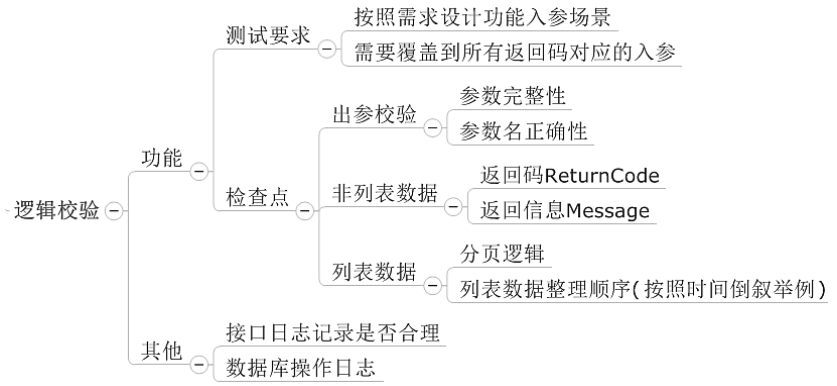


图 3-3-5　逻辑校验框架

3.3.5　接口测试用例设计——用例模板

作者参照业界一些比较好的用例设计模板，利用 Excel 表格设计了一个接口测试用例模板，如图 3-3-6 和图 3-3-7 所示。在表格中，除正常的用例元素外，增加了统计功能，以方便计算接口手工测试的通过率。

测试结果统计						
测试用例数	18					
BVT用例数	4					
A级用例数	10	PASS数	0			http:
B级用例数	4	FAIL数	0			
C级用例数	0	BLOCK数	0			
优先级	用例编号	功能描述	接口名称	前置条件	请求方式	
BVT	SN001	搜索内容	search_name	关键字有数据	GET	http://192.168.1.171:8081
A	SN002				GET	http://192.168.1.171:8081
A	SN003				GET	http://192.168.1.171:8081
B	SN004				GET	http://192.168.1.171:8081
BVT	SN005				GET	http://192.168.1.171:8081
A	SN006				GET	http://192.168.1.171:8081
B	SN007				GET	http://192.168.1.171:8081
A	SN008				GET	http://192.168.1.171:8081
BVT	SN009				GET	http://192.168.1.171:8081
A	SN010				GET	http://192.168.1.171:8081

图 3-3-6　用例模板（左半部分）

测试模块与说明					测试信息	
//192.168.1.171:8081/api/search_name					版本号	
					测试人员	
					测试时间	
接口地址	请求参数	返回包数据	预期结果	实际结果	测试结果	备注
/api/search_name?content=test						
/api/search_name?content=						
/api/search_name?content=NULL						
/api/search_name?content=''						
/api/search_name?content=testpiao						
/api/search_name?content=testpia						
/api/search_name?content=testpiao1						
/api/search_name?content=测试环境						
/api/search_name?content=测试						
/api/search_name?content='						
/api/search_name?content=@						
/api/search_name?content=,						
/api/search_name?content=;						
/api/search_name?content='1=1'						

图 3-3-7　用例模板（右半部分）

3.4　补充知识点

1．SQL 注入

SQL 注入是指，在不安全控件内输入一些 SQL 语句或其他数据库的语句，从而欺骗服务器执行，进行影响数据库中的数据。即在提交请求时，将请求数据替换成特殊的 SQL 语句，则服务器端在没有对数据做处理的情况下直接处理了请求数据（主要是在后台执行了 SQL 语句），从而得到非正常的返回结果。

2．SQL 注入防护

永远不要信任用户的输入。要对用户的输入进行校验，可以通过正则表达式、限制长度、对单引号和双（-）等来进行转换。

永远不要使用动态拼装 SQL。可以使用参数化的 SQL，或直接使用存储过程进行数据的查询与存取（3.3.2 小节所举中 SQL 语句采用的就是拼接方式）。

管理员权限可以操作数据库做很多事情，而一些应用程序只需要查询权限，故不需要管理员权限。

不要直接存放机密信息。应加密或者用 hash（哈希）算法对码和敏感的信息进行加密。

对应用程序的异常应给出尽可能少的提示信息。最好使用自定义的错误信息对原始错误信息进行包装。

检测 SQL 注入一般采取软件或网站平台：软件一般采用 SQL 注入检测工具 Jsky；网站平台一般采用亿思网站安全平台检测工具、MDCSOFT SCAN 等。

4 编程前的准备

本章讲什么：

1. Python 版本的选择；
2. Python 多版本共存；
3. 在 Windows 下安装 Python 和 MySQL。

工欲善其事，必先利其器。在正式讲解模块代码和编写代码之前，需要搭建最基本的 Python 环境，然后才是安装其他辅助软件和第三方模块等。

4.1 Python 环境准备

4.1.1 选择 Python 2 还是 Python 3

对于初学者应学 Python 2 还是 Python 3，一直有以下 3 种观点：

（1）学习 Python 2 或 Python 3 都可以，差别不大。学会其中一个，另外一个也很快就会了。

（2）学习 Python 2。因为 Python 的很多库都不兼容 Python 3，很多公司还是在使用 Python 2，往 Python 3 迁移的成本太高。

（3）学习 Python 3。Python 3 是趋势，其优势很明显，否则其开发人员是不会做如此大幅度的版本更新的。Python 2 到 2020 年就不被维护了。

针对以上 3 种观点，本人比较支持第（3）种，具体原因如下。

第（1）种观点显然是不负责任的。对初学者来说，随便选择一个版本学习，会使其在学习过程中总是产生疑惑和顾虑，不利于初学者坚定信念和勇往直前。初学者需要一个明确而肯定的答案。

第（2）种观点是比较陈旧的观点。以前的 Python 版本确实对 Python 3 的第三方模块支持不够。但现在早已升级了。况且，初学者本来就是一张白纸，没必要从老版本开始学习，这样后期还需要花费时间将版本升级到 Pyhton 3，费时又费力，所以不如一步到位直接选择学习 Pyhton 3。

所以，第（3）种观点也就不需要过多解释了。Python 3 之所以摒弃了之前的很多语法规则，就是为了让 Python 更完美。Python 3 比 Python 2 更容易学习，因为其抛弃了一些之前的“包袱”。并且，现在 Python 主流的库基本都已经完成了对 Python 3 的兼容，

例如：

- 核心库（NumPy、SciPy、Pandas）。
- 可视化（Matplotlib）。
- 机器学习（SciKit-Learn）。
- 自然语言处理（Gensim）。
- 数据挖掘（Scrapy）。
- Web 开发（Django、Flask）。

既然 Python 敢做如此大幅度的改版，直接使 Python 3 不向下兼容，说明其有充足的理由。

4.1.2　在 Windows 下安装 Python 3

（1）在 Python 官网中下载 Windows 版本的安装程序（官网地址：https://www.Python.org），如图 4-1-1 所示。

提示：如果要选择其他版本，则可以单击“Windows”选项，然后在打开的界面中选择相应的版本。

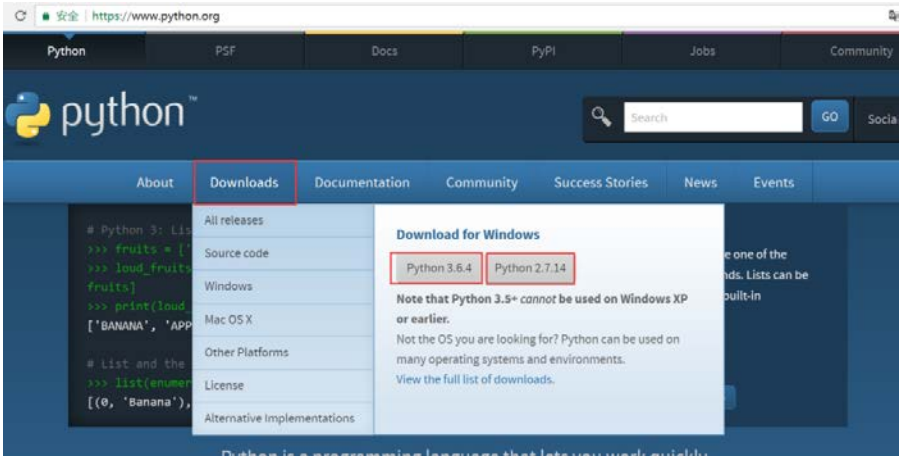


图 4-1-1　下载 Python 3 的安装程序

（2）在 Windows 操作系统中安装 Python 3，如图 4-1-2 和图 4-1-3 所示。

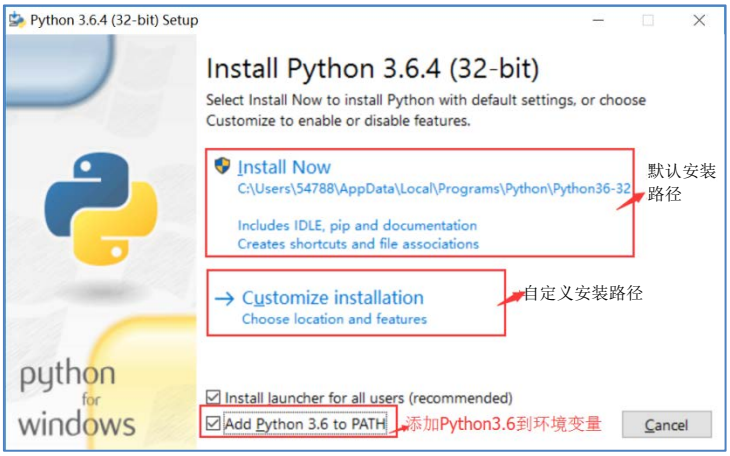


图 4-1-2　安装 Python 3 步骤 1

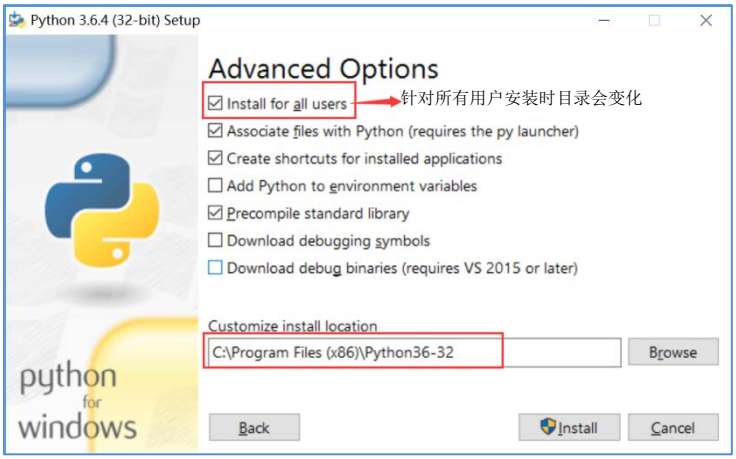


图 4-1-3　安装 Python 3 步骤 2

（3）在 DOS 窗口中执行 Python 命令，之后如果显示正常的版本信息和帮助信息，则说明安装成功。

4.1.3　Python 2 和 Python 3 共存之道

（1）将 Python 2 和 Python 3 分别安装在 C 盘中的同一个文件夹下，并分别命名为“Python 36”和“Python 27”，如图 4-1-4 所示。

图 4-1-4　本地双版本目录

（2）新增以下 4 个环境变量（如果已存在，则不需要再添加）：

- C:\Python27。
- C:\Python27\Scripts。
- C:\Python36。
- C:\Python36\Scripts。

（3）分别修改 Python27 和 Python36 的启动程序名称：

- 将 C:\Python27\Python.exe 修改为 C:\Python27\Python2.exe。
- 将 C:\Python36\Python.exe 修改为 C:\Python36\Python3.exe。

（4）分别在 Python2 和 Python3 文件夹下重新安装一下 pip。

```
Python2 -m pip install --upgrade pip --force-reinstall

Python3 -m pip install --upgrade pip --force-reinstall
```

（5）可以发现，默认的 pip 版本是 Python 2.7 下的，如图 4-1-5 所示。注意，这里的 V 是大写。

```
C:\Users\Administrator>pip2 -V
pip 9.0.1 from C:\Python27\lib\site-packages (python 2.7)

C:\Users\Administrator>pip3 -V
pip 9.0.1 from c:\python36\lib\site-packages (python 3.6)
```

图 4-1-5　双版本 pip

4.2　准备本地 MySQL 服务

（1）从 MySQL 官网中可以下载最新的 MySQL 安装文件（官网地址：https://dev.mysql.com/ downloads/mysql），如图 4-2-1 所示。

图 4-2-1　MySQL 下载页面

（2）根据自己的操作系统下载对应的 32 位或 64 位 MySQL 安装程序，如图 4-2-2 所示。

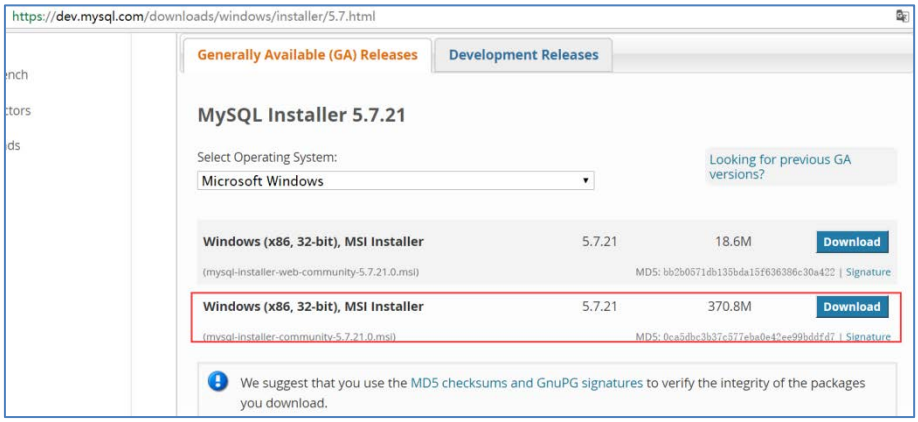


图 4-2-2　下载 MySQL 安装程序

（3）双击 MySQL 安装程序，在打开的对话框中进行设置，每一步设置完后单击“下一步”按钮，直到完成安装。请注意以下几个阶段的配置，如图 4-2-3 至图 4-2-5 所示。

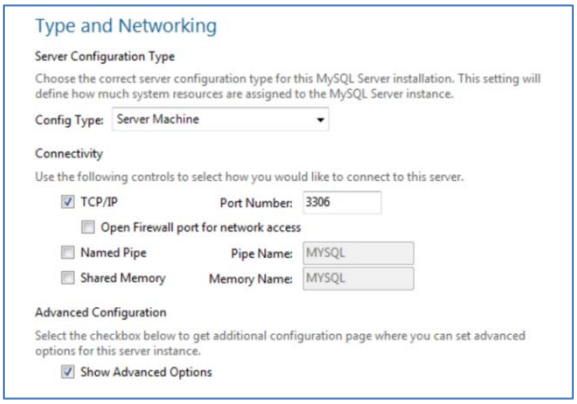


图 4-2-3　MySQL 安装过程 1

图 4-2-4　MySQL 安装过程 2

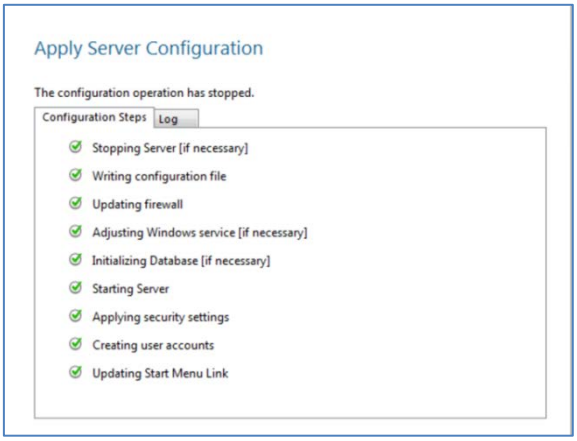


图 4-2-5　MySQL 安装过程 3

提示：

安装完 MySQL 后，要确保服务处于自启动状态，否则在后续使用过程中都要手动启动服务。具体方法是：在“我的电脑”图标上单击鼠标右键，在弹出的快捷菜单中选择“管理”命令，在打开的对话框中确认 MySQL 处于启动状态，如图 4-2-6 所示。

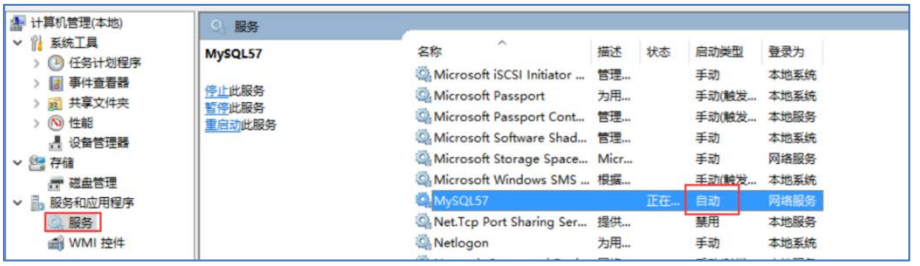


图 4-2-6　MySQL 处于启动状态

4.3　补充知识点

4.3.1　Python 2 与 Python 3 的语法区别

表 4-3-1 中列举的是 Python 2 和 Python 3 的主要区别。在实际使用中，两者之间的转换并不是特别费事，读者在使用时需要注意查看。

表 4-3-1　Python 2 与 Python 3 的主要区别

操作对象说明	Python 2 处理机制	Python 3 处理机制
获取字典键值对	iteritems()	items()
对中文的支持	输出中文时需要加 u，比如：u '中文'	直接把中文作为字符串，比如'中文'
除法	带上小数点时“/”时表示真除，如 1/2=0，1.0/2=0.5	“/”表示真除，如 1/2=0.5
输出	print " "	print()
等待输入函数	raw_input	input
判断字典是否存在某个键	dict.has_key(key)	key in dict:
数据类型	Python 2 中有 long 类型数据	Python 3 中全部改成了 int 类型，取消了 long 类型

续表

操作对象说明	Python 2 处理机制	Python 3 处理机制
比较	两个参数类型不同时，随机返回 bool 值	两个参数类型不同时，抛出 TypeError 异常信息
bool 值	0，1	True，False
异常处理	Try…except Exception, e :	Try…except Exception as e :

4.3.2 Python 解释器

在执行 Python 代码的过程中，解释器起着“翻译”的作用，它把 Python 代码编译成能被机器识别的内容，如图 4-3-1 所示。Python 有各种不同的解释器，但最常用的是 CPython。

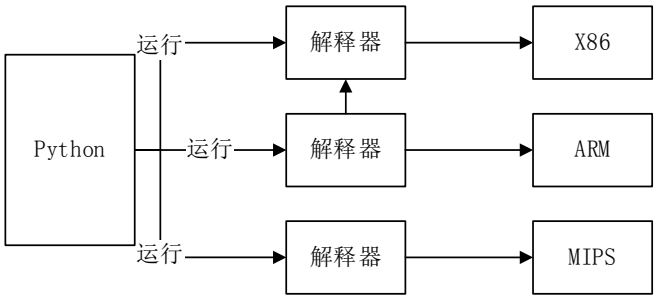


图 4-3-1 Python 解释器的作用

下载并安装好 Python 3 后，就直接获得了一个 Python 官方版本的解释器——CPython，这个解释器是用 C 语言开发的。在命令行下运行 Python，就启动了 CPython 解释器。

CPython 是使用最广的 Python 解释器。本书的所有代码都可以在 CPython 下运行。

4.3.3 Python 的函数

Python 的函数用于接收参数，并返回函数执行后的结果。其参数个数为 0~N，接口是必须要有的，否则返回的是 None。

Python 函数定义：使用 def 来标志一个函数，def 后面紧跟的是函数名。下面的代码定义了一个简单的函数，其功能是判断某个数为正或为负。

其中，函数名称是 function，函数名称后面的小括号中是函数的参数名称，其中允许有 0 到多个参数，参数之间以英文逗号分开。函数的最后使用 return，即返回一个结果。

```
# -*- coding: utf-8 -*-
#__author__ = '大婶 N72'
#Python 函数
def function(data):
    if data>=0:
        print("正数")
        return data
    else:
        print("负数")
        return -data

result=function(1)
print(result)
```

函数允许一次返回多个结果。在返回多个结果时，可以使用多个参数来承接函数值，也可以使用一个参数来承接函数值。之所以可以使用一个参数来承接函数值，是因为函数返回的结果是一个元组。读者可以运行下面的代码试一试，要重点看一下承接参数的区别。

```
# -*- coding: utf-8 -*-
#__author__ = '大婶 N72'
#Python 函数
def function(data):
    if data>=0:
        return data,data
    else:
        return data,-data

original,result =function(-1)
print(original,result)
result =function(-1)
print(result[0])
```

5

用 Python 操作 MySQL 数据库

本章讲什么：

1. 对 Python 处理 MySQL 数据库的代码进行分析；
2. 本章涉及的 Python 部分语法；
3. PyCharm 的使用。

目前 Python 的版本主要有两个——Python 2 和 Python 3。这两个版本的语法区别还是比较大的。如果你是初学 Python，则建议你直接从 Python 3 学起。本书所有代码都是基于 Python 3 版本的。

5.1 提前工作

想要用 Python 操作 MySQL 数据库，首先需要确认以下工作是否完成。

（1）搭建 Python 3 环境。

（2）搭建 MySQL 数据库连接工具（Navicat Premium，基于 Windows 平台）。

（3）搭建 MySQL 服务器（在本地或者远程服务器中都可以）。通过连接工具查看是否能成功连接 MySQL 服务器，如图 5-1-1 和图 5-1-2 所示。

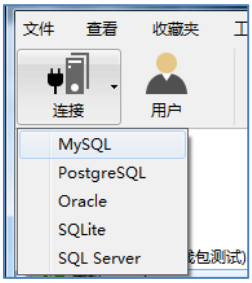


图 5-1-1　连接 MySQL 服务器

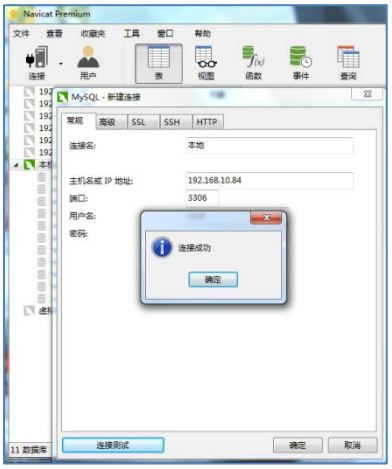


图 5-1-2　成功连接 MySQL 服务器

（4）新建数据库。在新连接的数据名称上单击鼠标右键，在弹出的快捷菜单中选择“新建数据库”命令。在打开的对话框中将数据库名设为“test_interface”，具体设置如图 5-1-3 所示。

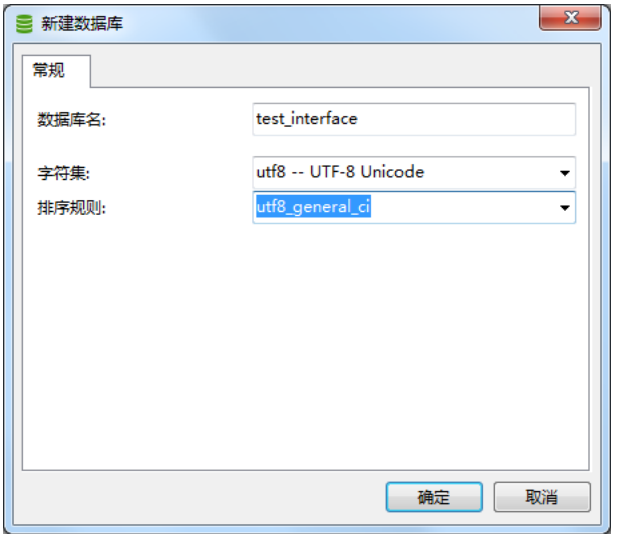


图 5-1-3　新建数据库

（5）新建数据库表并初始化测试数据。

下面的代码为在 MySQL 数据库中创建配置表（config_total）并新增一条配置信息。

```
CREATE TABLE 'config_total' (
  'id' int(2) NOT NULL AUTO_INCREMENT,
  'key_config' varchar(128) DEFAULT NULL COMMENT '关键字名称',
  'value_config' text COMMENT '关键字值',
  'description' varchar(128) DEFAULT NULL COMMENT '关键字解释信息',
  'status' int(2) DEFAULT NULL COMMENT '配置文件状态，1—有效，0—无效',
  'create_time' timestamp NULL DEFAULT CURRENT_TIMESTAMP COMMENT '创建时间
',
  'update_time'  timestamp  NULL  DEFAULT  CURRENT_TIMESTAMP  ON  UPDATE
CURRENT_TIMESTAMP COMMENT '更新时间',
  PRIMARY KEY ('id')
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8 COMMENT='接口测试配置
表';

INSERT INTO 'config_total' ('key_config', 'value_config', 'description',
'status') VALUES ('test', 'value_test', '测试配置', '1');
```

如果以上工作都已经完成了，那么恭喜你，下面可以开始编写和调试代码了。

提示：

如果在创建数据库表时提示“[Error] 1293 - Incorrect table definition; there can be only one TIMESTAMP column with CURRENT_TIMESTAMP in DEFAULT or ON UPDATE clause”，那么这是因为 MySQL 服务器的版本在 5.6 以下，这时可以通过 Navicat Premium 手动创建表，或者将 MySQL 服务器的版本升级到 5.6 以上。

5.2 操作 MySQL 数据库

5.2.1 用 Python 操作 MySQL 数据库的流程

图 5-2-1 所示的是用 Python 操作 MySQL 数据库的流程。

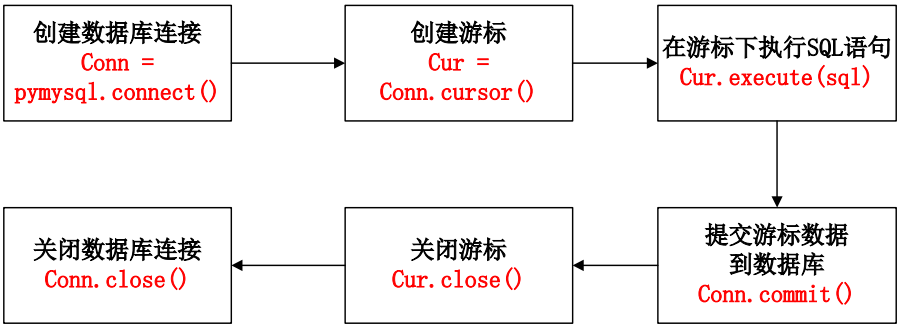


图 5-2-1 用 Python 操作 MySQL 数据库的流程

对于该流程的解释如下。

（1）创建数据库连接：即和数据库建立连接。这样后续的操作才能被传递到数据库服务器中。创建数据库连接就是建立一个通道。

（2）创建游标：即将受影响的数据暂时存放到一个内存区域的虚表中，而这个虚表就是游标。为什么要建立游标呢？目的是为了回滚。此时将对数据库的操作暂时存放在游标中，只要不提交，就可以对游标中的内容进行回滚。

（3）在游标下执行 SQL 语句：此时会将执行结果存放在游标中。

（4）提交游标数据到数据库：这一步才是真正把游标中的数据更新到数据库中，如果缺少这一步，那么即使执行了 SQL 语句，数据库中的数据也不会有变化。

（5）关闭游标。

（6）关闭数据库连接：要养成每次用完数据库后都关闭数据库连接的习惯，因为在 Python 中建立数据库连接后会占用解释器的线程，如果不关闭数据库连接，则会使其他正在运行的程序变慢。

5.2.2　用 Python 操作 MySQL 代码

1．新建 Python 工程及其文件

可以按照下面的方法新建一个 opmysql.py 文件。

方法一：新建 txt 文件，之后重命名为 opmysql.py。该方法适合独立的代码。

方法二：借助 PyCharm 新建工程，在新建的工程下新建 opmysql.py 代码文件（推荐使用该方法）。具体步骤如下所示。

（1）选择菜单中的“file”→“New Project”命令，在弹出的对话框中选择“Pure Python”选项，如图 5-2-2 所示。

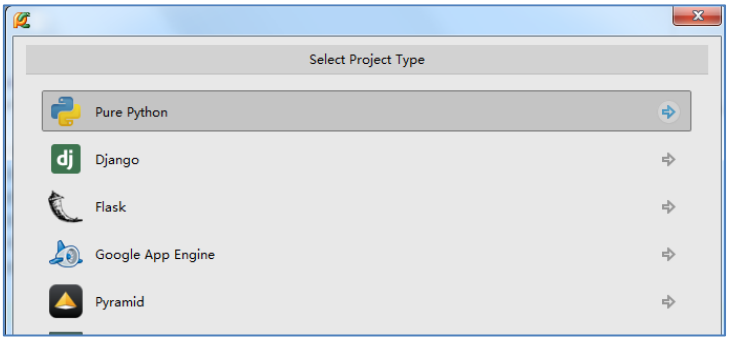


图 5-2-2　新建工程

（2）出现如图 5-2-3 所示的界面。其中 Location 表示存放新建工程的文件夹；

Interpreter 表示该工程下的 py 文件使用哪个目录下的 Python.exe 文件，一般选择在本地安装的 Python 目录下的 Python.exe 文件（这里选择 Python3.exe 文件）。

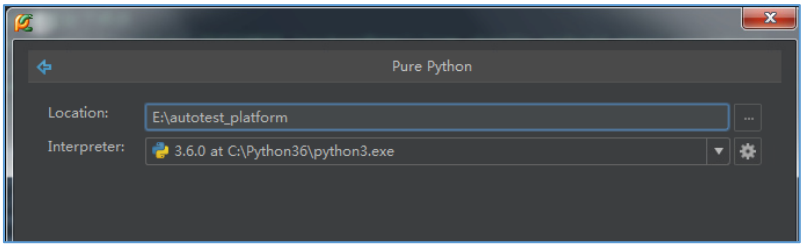


图 5-2-3　新建工程名

（3）选择打开工程的方式，如图 5-2-4 所示。

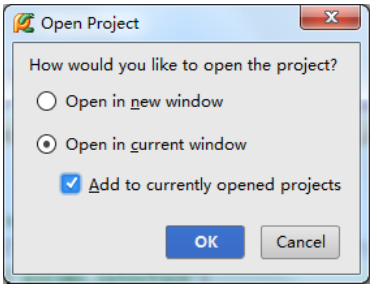


图 5-2-4　选择打开工程的方式

- Open in new window：在新的 PyCharm 窗口中打开这个工程。选择该项后，会新打开一个 PyCharm 窗口。
- Open in current window：替换当前已经打开的工程，在当前窗口中打开工程。
- Add to currently opened projects：在工程列表中新增一个工程，不覆盖之前已打开的工程（推荐使用该方法）。

（4）在 PyCharm 左侧工程列表中，在新建的工程名上单击鼠标右键，在弹出的快捷菜单中选择“New”→“Python Package”命令，如图 5-2-5 所示。新建一个 Python 包，将其命名为“common”。

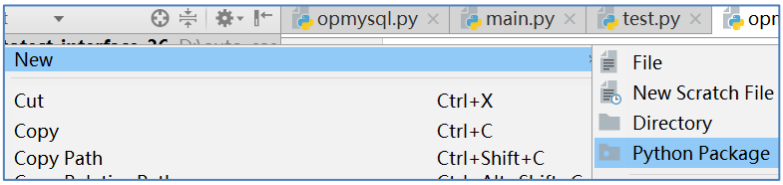


图 5-2-5　新建一个 Python 包

（5）在 common 包上单击鼠标右键，在弹出的快捷菜单中选择“New”→“Python file”命令，新建 py 文件。在弹出的对话框中将“Name”设为“opmysql”，如图 5-2-6 所示。

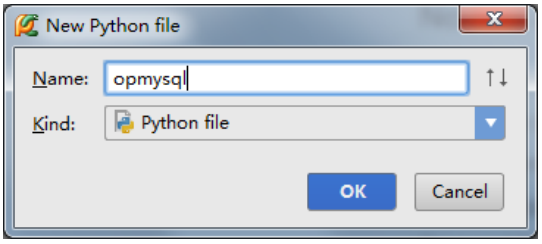


图 5-2-6　设置 Name

（6）单击“OK”按钮后，成功新建工程及其目录下的 py 文件，目录结构如图 5-2-7 所示。

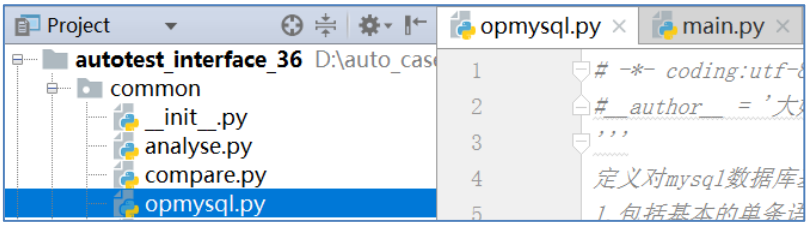


图 5-2-7　新建 py 文件

新建一个 config.py 文件，并初始化数据。

（1）在工程目录下新建一个 public 包。

（2）在 public 目录下新建一个 config.py 文件。

（3）在 config.py 文件中编写以下代码。

```
# -*- coding:utf-8 -*-
#__author__ = '大婶 N72'
import os
src_path = os.path.dirname(os.path.dirname(os.path.realpath(__file__)))#
当前代码所在目录的上级目录
```

2．在 py 文件下新建代码

代码：opmysql.py（以下只列出了主要的行）

```
1  # -*- coding:utf-8 -*-
2  #__author__ = '大婶 N72'
3  '''
4  定义对 MySQL 数据库基本操作的封装
5  1.包括基本的单条语句操作，如删除、修改、更新
6  2.独立地查询单条、多条数据
7  3.独立地添加多条数据
8  '''
9  import logging,os,pymysql
10  from public import config
11
12  class OperationDbInterface(object):
13      #初始化数据库连接
14              def  __init__(self,host_db='192.168.0.103',user_db='root',
passwd_db='root',
name_db='test_interface',port_db=3306,link_type=0):...
……
37      #定义单条数据操作，包含删除、更新操作
38      def op_sql(self,condition):...
……
56      #查询表中单条数据
57      def select_one(self, condition):...
……
78      #查询表中多条数据
79      def select_all(self,condition):...
……
100      #定义表中插入数据操作
101      def insert_data(self,condition,params):...
……
119      #关闭数据库
120      def __del__(self):...
```

```
……
126  if __name__ == "__main__":
127      test=OperationDbInterface()#实例化类
128      result_select_all=test.select_all("SELECT * FROM config_total")#
查询多条数据
129      result_select_one=test.select_one("SELECT * FROM config_total WHERE
id=1")#查询单条数据
130                  result_op_sql=test.op_sql("update   config_total   set
value_config='test' WHERE id=1")#通用操作
131                               result=test.insert_data("insert     into
config_total(key_config,value_config,description,status)           values
(%s,%s,%s,%s)",[('mytest1','mytest11','我的测试 1',1),('mytest2','mytest22',
'我的测试 2',0)])#插入操作
132      print(result_select_all['data'], result_select_all['message'])
133      print(result_select_one['data'], result_select_one['message'])
134      print(result_op_sql['data'], result_op_sql['message'])
135      print(result['data'], result['message'])
136      # if result['code']=='0000':
137      #     print(result['data'],result['message'])
138      # else:
139      #     print(result['message'])
```

上面的代码是 opmysql.py 文件中的一部分，从中大致能看出此方法的结构。具体在每个方法中怎么实现，会在下面逐步讲解。在这里读者可以先了解一下整体的结构。

该代码是一个封装了 Python 在操作 MySQL 数据库时常用方法的类。这个类的类名是 OperationDbInterface，此类下有 6 个方法。

- __init__：初始化数据库连接。
- op_sql：通用操作方法。
- select_one：查询单条数据。
- select_all：查询多条数据。
- insert_data：添加数据。
- __del__：关闭数据库连接。

读者也可以根据实际的需要新增或合并对应的方法。总体的设计思想是：将常用的

方法预先写好，在需要时直接调用。

最后是“if _ _name_ _ == "_ _main_ _"”，其意思是，让代码模块既可以被导入别的模块中使用，也可以在该判断条件下执行。那么该条件判断的具体作用是什么呢?

（1）如果直接执行某个.py 文件，比如这里直接执行 opmysql.py 文件，该文件中的“_ _name_ _ == '_ _main_ _'”是 True，则会执行其下面的代码，如图 5-2-8 所示。

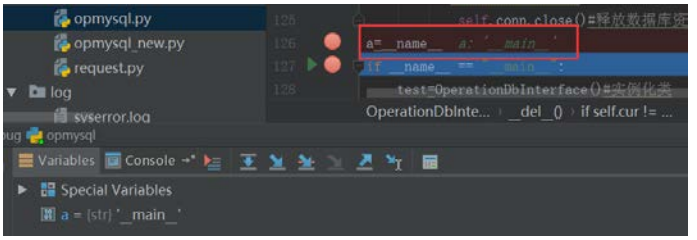


图 5-2-8　执行 py 文件时的判断条件

（2）如果是从另外一个.py 文件通过 import 导入该文件，那么这时“_ _name_ _”的值就是这个 py 文件的名字，而不是“_ _main_ _”（即“_ _name_ _ == '_ _main_ _'”是 False），也就不会执行下面的代码。

（3）在调试代码时，在“if _ _name_ _ == '_ _main_ _'”中加入了一些调试代码，这些代码在调用外部模块时不执行。但在排查问题时，执行该模块后，则调试代码能正常执行。所以，在 if 语句后面写入的是实例化类和调用类中的方法。

下面是该 py 文件的全部代码。

代码：opmysql.py

```
# -*- coding:utf-8 -*-
#__author__ = '大婶 N72'
'''
定义对 MySQL 数据库基本操作的封装
1.包括基本的单条语句操作，如删除、修改、更新
2.独立地查询单条、多条数据
3.独立地添加多条数据
'''
```

```
import logging,os,pymysql
from public import config

class OperationDbInterface(object):
    #定义初始化数据库连接
      def __init__(self,host_db='192.168.0.104',user_db='root',passwd_db='root',name_db='test_interface',port_db=3306,link_type=0):
        """
        :param host_db: 数据库服务主机
        :param user_db: 数据库用户名
        :param passwd_db: 数据库密码
        :param name_db: 数据库名称
        :param port_db: 端口号，整型数据
        :param link_type: 连接类型，用于设置输出数据是元组还是字典，默认是字典，link_type=0
        :return:游标
        """
        try:
           if link_type == 0:
                  self.conn=pymysql.connect(host=host_db,user=user_db, passwd=passwd_db, db=name_db, port=port_db,
                                            charset='utf8',cursorclass = pymysql.cursors.DictCursor)#创建数据库连接，返回字典
           else:
               self.conn = pymysql.connect(host=host_db, user=user_db, passwd=passwd_db, db=name_db, port=port_db,
                                           charset='utf8')  # 创建数据库连接，返回元组
           self.cur=self.conn.cursor()
        except pymysql.Error as e:
             print("创建数据库连接失败|Mysql Error %d: %s" % (e.args[0], e.args[1]))
                    logging.basicConfig(filename  =  config.src_path  + '/log/syserror.log',level                                                    = logging.DEBUG,format='%(asctime)s %(filename)s[line:%(lineno)d] %(levelname)s %(message)s')
           logger = logging.getLogger(__name__)
           logger.exception(e)
```

下面是对上一段代码的解释。

- 第 1~8 行代码：Python 以“#”作为单条语句的屏蔽符号，以一对“""”符号作为多条语句的屏蔽符号。第 1~8 行是作者的备注信息和当前 py 文件的功能解释。在第 1 行中，“coding:utf-8”是编码声明，其作用是让代码支持中文，比如注释信息等。
- 第 9、10 行代码：导入需要的模块。模块是 Python 的强大之处，即将别人写好的一个功能导入之后就可以直接使用。也可以修改自己在本地的模块，或者自己写模块，但是这些模块在被使用前都需要先导入（import）。这两行代码的作用是导入 logging、os 和 pymysql 三个模块。而这三个模块分别是日志模块、操作系统模块、MySQL 数据库模块。其中，pymysql 是配合 Python 3 版本使用的，需要另行安装。logging 和 os 是系统自带的，无须另行安装。
- 第 12 行代码：定义一个类，其名称为 OperationDbInterface，其继承的父类默认使用 object 超类（前提是该类没有父类）。
- 第 14 行代码：在类的下面使用_ _init_ _()方法来初始化数据（注意，init 前后分别有两条下画线），并创建数据库连接和游标。方法的默认参数是 self，self 是类下面每个方法中的第一个参数，且是必需的一个参数。其后的 host_db 代表数据库服务器主机地址（如果 MySQL 服务在本机上，就是本地的 IP）。user_db 代表数据库的账号，passwd_db 代表数据库的密码，name_db 代表数据库的名称。port_db 代表数据库的端口号（注意端口号是数字，默认端口号是 3306）。link_type 代表返回数据的格式，默认值是 0。在这个_ _init_ _ ()方法中，每个参数都被事先设置了默认值，当其他代码使用这个类时，在默认情况下无须再传递这些参数。如果参数值有变化，则只需要传递对应的参数及其值即可。
- 第 24~33 行代码：其结构是 try…except 异常处理结构体。第 26 行代码按照_ _init_ _()方法中传递的参数值来建立数据库连接，依据 link_type 值返回不同的结果，这里分别是返回字典的游标和元组的游标。在第 32、33 行代码中，except 捕获到所有 pymysql.Error 错误信息，并使用 as 将其重命名给参数 e。第 33 行代码用于输出错误信息中的两个参数。这里先关闭数据库服务再进行代码测试。在该情况下，日志信息如图 5-2-9 所示。

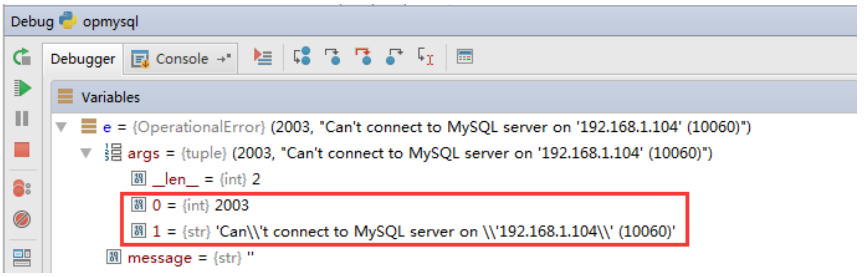


图 5-2-9　日志信息

最终打印信息如图 5-2-10 所示。

```
创建数据库连接失败|Mysql Error 2003: Can't connect to MySQL server on '192.168.1.104' (10060)
```

图 5-2-10　最终打印信息

- 第 34~36 行代码：系统的日志记录。需要注意的是，日志文件的默认存储路径为.../log/syserror.log，需要将其修改为自己所需的路径，文件名称是 syserror.log，目录是当前这个 opmysql.py 文件上级目录下的 log 文件夹，如图 5-2-11 所示。

```
2017-12-12 10:35:41,608 opmysql.py[line:31] ERROR (2003, "Can't connect to MySQL server on '192.168.1.104' (10060)")
Traceback (most recent call last):
  File "F:/Autotest_interface_demo/common/opmysql.py", line 22, in __init__
    self.conn=MySQLdb.connect(host=host_db,user=user_db, passwd=passwd_db, db=name_db, port=port_db, charset='utf8')#创建数据库链接
  File "C:\Python27\lib\site-packages\MySQLdb\__init__.py", line 81, in Connect
    return Connection(*args, **kwargs)
  File "C:\Python27\lib\site-packages\MySQLdb\connections.py", line 187, in __init__
    super(Connection, self).__init__(*args, **kwargs2)
OperationalError: (2003, "Can't connect to MySQL server on '192.168.1.104' (10060)")
```

图 5-2-11　系统的日志记录

代码：opmysql.py

```
37      #定义单条数据操作，包含删除、更新操作
38      def op_sql(self,condition):
39          """
40          :param condition: SQL 语句，该通用方法可用来替代 updateone，deleteone
41          :return:字典形式
42          """
43          try:
44              self.cur.execute(condition)#执行 SQL 语句
45              self.conn.commit()#提交游标数据
```

```
46            result={'code':'0000','message':'执行通用操作成功','data':[]}
47        except pymysql.Error as e:
48            self.conn.rollback()  # 执行回滚操作
49            result={'code':'9999','message':'执行通用操作异常','data':[]}
50            print("数据库错误|op_sql %d: %s" % (e.args[0], e.args[1]))
51                    logging.basicConfig(filename  =  config.src_path  +
'/log/syserror.log', level = logging.DEBUG, format='%(asctime)s %(filename)
s[line:%(lineno)d] %(levelname)s %(message)s')
52            logger = logging.getLogger(__name__)
53            logger.exception(e)
54        return result
```

下面是对上一段代码的解释。

第 37~54 行代码：定义了通用的单条数据操作方法，用于替代两种操作——更新单条数据和删除单条数据。也可以将该方法拆分为两个方法独立使用。该方法名为 op_sql，参数只有一个 condition，其接收的是字符串类型的 SQL 语句。

- 第 44 行代码：用于执行这个 SQL 语句，语句执行完成后受影响的数据会被存放在游标中，此时并没有影响到实际的数据库。只有在执行了第 45 行代码（提交游标数据到数据库），才真正影响到数据库。
- 第 46 行代码：执行完成后返回的结果是一个字典，这里参照了一般接口返回数据的格式来定义，字典包含了 3 个参数。

  code 参数及其值：用来表示这次执行的结果。在正常情况下，如果该参数值为 0000，则代表执行成功；如果为其他值，则有对应的返回码。

  message 参数及其值：作为执行情况的描述信息。

  data 参数及其值是辅助参数：在本方法中都是空列表，这是因为需要临时占位，如果后期需要返回更多的信息，则可以通过该字段来补充。但在 select 类型的方法中，它作为查询结果的数据。
- 第 48 行代码：执行异常时的回滚操作。
- 第 49 行代码：和第 46 行代码类似结构，是在失败情况下的返回字典。
- 第 54 行代码：使用 return 来统一返回 result 参数，作为这个方法统一的返回结果，无论其处理过程是正常的还是异常的。

代码：opmysql.py

```
56      #查询表中单条数据
57      def select_one(self, condition):
58          """
59          :param condition: SQL 语句
60          :return: 字典形式的单条查询结果
61          """
62          try:
63              rows_affect = self.cur.execute(condition)
64              if rows_affect > 0:  # 查询结果返回数据数大于 0
65                  results = self.cur.fetchone()  # 获取一条结果
66                  result = {'code': '0000', 'message': '执行单条查询操作成功',
'data': results}
67              else:
68                  result = {'code': '0000', 'message': '执行单条查询操作成功',
'data': []}
69          except pymysql.Error as e:
70              self.conn.rollback()  # 执行回滚操作
71              result = {'code': '9999', 'message': '执行单条查询异常', 'data':
[]}
72              print("数据库错误|select_one %d: %s" % (e.args[0], e.args[1]))
73                      logging.basicConfig(filename  =  config.src_path  +
'/log/syserror.log',
level=logging.DEBUG,format='%(asctime)s %(filename)s[line:%(lineno)d] %(l
evelname)s %(message)s')
74              logger = logging.getLogger(__name__)
75              logger.exception(e)
76          return result
```

下面是对上一段代码的解释。

第 56~76 行代码：定义查询单条数据操作的方法。由于需要得到查询结果，所以该方法单独编写。

- 第 63 行代码：用来判断受影响的数据量，即按照条件是否查询到数据，以及有无数据，返回的结果中有不同的 data 数据。
- 第 65 行代码：使用 fetchone()方法获取唯一的数据。建议在使用该方法时，condition 参数执行的结果是唯一的。如果有多条数据，则按照返回数据只取一条。

- 第 66 行代码：与前面方法中返回结果的不同之处在于 data 数据。由于是查询语句，所以需要得到查询的数据。
- 第 68 行代码：在没有查询到数据时，data 是空列表。调用方依据这个值可以做进一步的处理。

代码：opmysql.py

```
 77     #查询表中多条数据
 78     def select_all(self,condition):
 79         '''
 80         :param condition: SQL 语句
 81         :return:字典形式的批量查询结果
 82         '''
 83         try:
 84             rows_affect=self.cur.execute(condition)
 85             if rows_affect>0:        #查询结果返回数据数大于 0
 86                 self.cur.scroll(0, mode='absolute') # 将鼠标光标放回到初始位
置
 87                 results = self.cur.fetchall()       # 返回游标中所有结果
 88                     result={'code':'0000','message':'执行批量查询操作成功
','data':results}
 89             else:
 90                     result={'code':'0000','message':'执行批量查询操作成功
','data':[]}
 91         except pymysql.Error as e:
 92             self.conn.rollback()  # 执行回滚操作
 93             result={'code':'9999','message':'执行批量查询异常','data':[]}
 94             print("数据库错误|select_all %d: %s" % (e.args[0], e.args[1]))
 95                       logging.basicConfig(filename  =  config.src_path  +
'/log/syserror.log',            level           =           logging.DEBUG,
format='%(asctime)s %(filename)s[line:%(lineno)d] %(levelname)s %(message
)s')
 96             logger = logging.getLogger(__name__)
 97             logger.exception(e)
 98         return result
```

下面是对上一段代码的解释。

第 77~98 行代码：定义查询多条数据操作的方法。该方法与 select_one 方法不同之

处在于，其获取数据是从游标开始位置算起的，然后使用 fetchall()方法获取所有数据（详见第 86~88 行代码）。

代码：opmysql.py

```
99     #定义表中插入数据操作的方法
100     def insert_data(self,condition,params):
101         '''
102         :param condition: insert 语句
103            :param params: insert 数据，列表形式[('3','Tom','1 year 1 class','6'),('3','Jack','2 year 1 class','7'),]
104        :return:字典形式的批量插入数据结果
105         '''
106         try:
107             results=self.cur.executemany(condition,params)#返回插入的数据条数
108             self.conn.commit()
109                 result={'code':'0000','message':'执行批量查询操作成功','data':results}
110         except pymysql.Error as e:
111             self.conn.rollback()  # 执行回滚操作
112                    result={'code':'9999','message':'执行批量插入异常','data':[]}
113                 print("数据库错误|insert_more %d: %s" % (e.args[0], e.args[1]))
114                     logging.basicConfig(filename = config.src_path + '/log/syserror.log',           level           =           logging.DEBUG, format='%(asctime)s %(filename)s[line:%(lineno)d] %(levelname)s %(message)s')
115             logger = logging.getLogger(__name__)
116             logger.exception(e)
117         return result
```

上一段代码定义表中插入数据操作的方法。该方法需要传递两个参数：condition 参数是要执行的插入（insert）SQL 语句；params 参数是要插入的具体数据（其形式为列表，列表中的数据是元组形式，列表中的每个数据对应 insert 中的参数）。在插入数据时，我们比较关心的是插入的数据条数是否正确，所以第 107 行代码使用 executemany 方法来执行插入多条数据操作，同时其返回值是插入成功的条数，将其赋值给 results，作为

整个方法返回结果的 data 值。

代码：opmysql.py

```
118     #关闭数据库
119     def __del__(self):
120         if self.cur != None:
121             self.cur.close()       #关闭游标
122         if self.conn != None:
123             self.conn.close()      #释放数据库资源
124
125 if __name__ == "__main__":
126     test=OperationDbInterface()   #实例化类
127      result_select_all=test.select_all("SELECT * FROM config_total")#
查询多条数据
128     result_select_one=test.select_one("SELECT * FROM config_total WHERE
id=1")#查询单条数据
129                 result_op_sql=test.op_sql("update   config_total   set
value_config='test' WHERE id=1")#通用操作
130                               result=test.insert_data("insert     into
config_total(key_config,value_config,description,status)           values
(%s,%s,%s,%s)",[('mytest1','mytest11',' 我 的 测 试  1',1),('mytest2','
mytest22','我的测试 2',0)])#插入操作
131     print(result_select_all['data'], result_select_all['message'])
132     print(result_select_one['data'], result_select_one['message'])
133     print(result_op_sql['data'], result_op_sql['message'])
134     print(result['data'], result['message'])
135     # if result['code']=='0000':
136     #     print(result['data'],result['message'])
137     # else:
138     #     print(result['message'])
```

下面是对上一段代码的解释。

第 118~123 行代码：定义关闭数据库连接操作的方法。这里使用的是_ _del_ _()方法，该方法是专用的删除操作方法，与_ _init_ _初始化方法类似。该方法主要用于完成两个操作，一是关闭游标，二是关闭数据库连接。

- 第 125~138 行代码：在 if 判断下的调试代码。在当前 opmysql.py 文件中，它主要用来测试类中方法的逻辑是否正确，也可以将其理解为单元测试代码。
- 第 126 行代码：实例化定义的类，实例化的名称是 test。
- 第 127~134 行代码：使用实例化下的方法做对应的测试，检查返回结果是否正确。由于每个方法都有返回结果，所以在该处可以通过打印 result 中的 code 情况，判断方法是否按照预期的逻辑执行。
- 第 135~138 行代码：体现了以上说法，正确时返回 data 值，异常时返回 message 值。

5.3　本章所涉及的 Python 语法

5.3.1　模块与包

Python 的程序由包（package）、模块（module）、函数和类组成。包是由一系列模块组成的集合，模块是处理某一类问题的函数和类的集合，如图 5-3-1 所示。

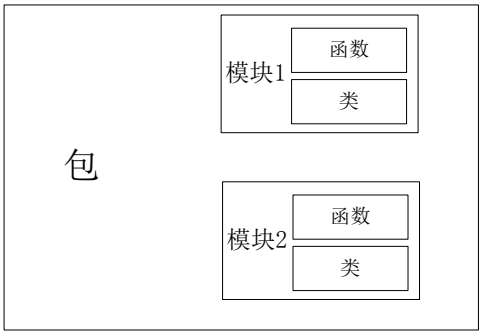


图 5-3-1　Python 中的包、模块、函数和类的关系

1．模块与包的使用

包是一个完成特定任务的工具箱。在 Python 中提供了许多有用的工具包，如字符串处理、图形用户接口、Web 应用程序、图形图像处理等。Python 自带的工具包和模块，默认被安装在 Python 的安装目录下的 Lib\site-packages 子目录中，如图 5-3-2 所示。在

py 文件中，在使用任何包和模块前都需要将其导入，比如在前面的代码中用到的 pymysql 模块，如图 5-3-3 所示。

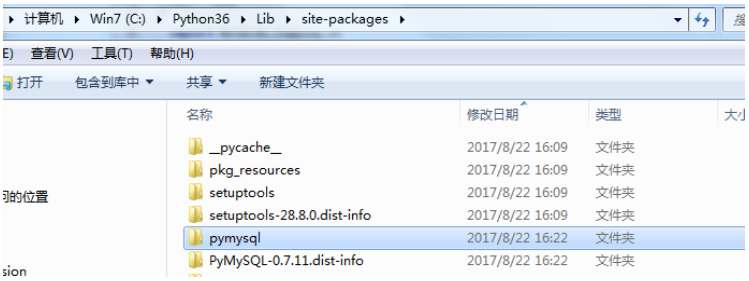


图 5-3-2　导入模块与包

图 5-3-3　Python 模块与包的使用

2．模块与包的基础

在 Python 中，一个.py 文件被称为一个模块（Module）。

（1）为什么要用模块？

模板其实是代码的封装和共用。不可能所有的代码都能被写在一个文件里面，所以会有很多个.py 文件。那么它们是怎么互相引用的呢？答案就是导入.py 文件，而这些被导入的文件被称为模块。

（2）模块的层级关系。

模块的上一级是包（Package），包的上一级还是包。包下面有一个_ _init_ _.py 文件，如图 3-3-4 所示。

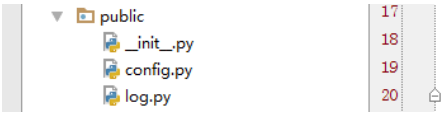


图 5-3-4　Python 包的结构

（3）引用模块的方法。

引用模块的方法有如下三种。

① from *包名* import *模块名*

在代码中可以直接使用模块名。比如，在其他 py 文件中使用本章的 opmysql.py 文件中的代码，如图 5-3-5 所示。

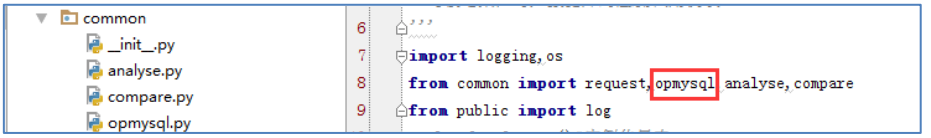


图 5-3-5　引用 Python 模块①

② import *包名*

在代码中引用模块时需要加“.”号，即“包名.模块名”，多级包目录就需要加多个“.”号。比如，在本章代码中使用了 pymysql 包下的模块，如图 5-3-6 所示。

图 5-3-6　引用 Python 模块②

③ import *模块名*

在代码中可以直接引用模块名。调用该模块的函数或类时，需要以模块名作为前缀，比如“模块名.函数名”，如图 5-3-7 所示。

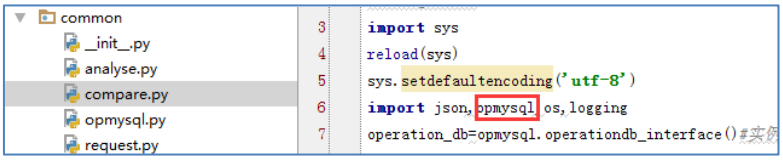


图 5-3-7　引用 Python 模块③

（4）模块别名。

如果用“import *模块名* as *别名*”方式来指定模块别名，那么后续代码中就使用“别名”来代替模块名。可以在运行代码时，根据当前环境选择最合适的模块，比如，Python 标准库一般会提供 StringIO 和 cStringIO 两个库，这两个库的接口和功能是一样的。但是，cStringIO 是用 C 语言编写的，速度更快，所以我们经常会看到以下写法：

```
try:
    import cStringIO as StringIO
    except ImportError: #导入失败会捕获到 ImportError
    import StringIO
```

这样可以优先导入 cStringIO。如果有些平台不提供 cStringIO，则可以降级使用 StringIO。在导入 cStringIO 时，用 import … as … 指定了别名为 StringIO，因此，后续代码引用 StringIO 即可正常工作。

3．包与模块的安装

常用的安装方法有以下 6 种：

（1）执行 pip install package 命令。

（2）执行 easy_install package 命令。

（3）使用 PyCharm 工具安装包。具体方法是：在 PyCharm 中，选择菜单中的“File”→“Settings”→“Project Interpreter”命令，在右侧窗口中单击“+”按钮，然后输入要安装的第三方包的名称并安装，如图 5-3-8 所示。

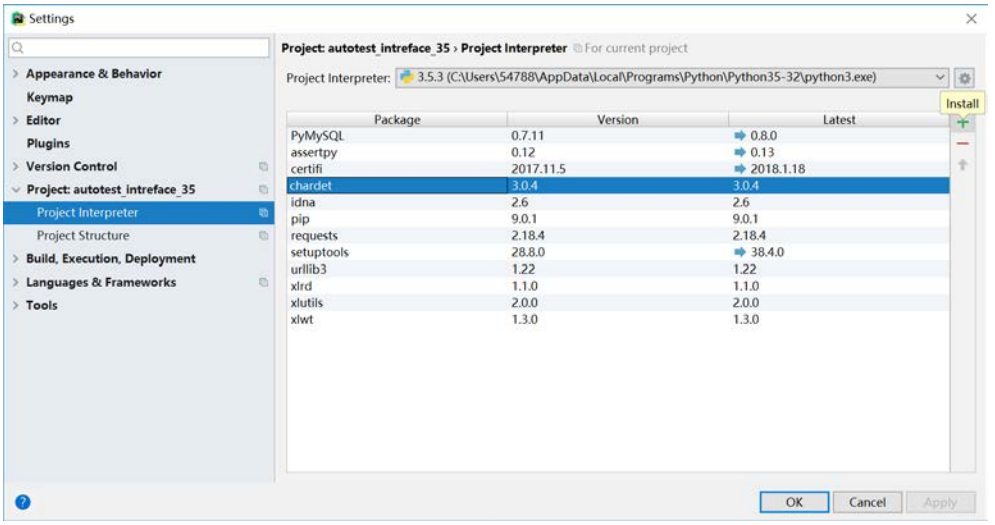


图 5-3-8　安装第三方包

（4）下载带有.exe 后缀的包，然后安装。

（5）下载模块的压缩包并解压缩，在解压缩目录下执行 Python setup.py install 命令。

（6）复制现成的模块包，将其放置在 Lib\site-packages 目录下。如果可以正常导入，则说明可以安装。

下面两个地址是包与模块的安装文件路径：

- https://pypi.Python.org/pypi
- https://sourceforge.net/directory/development

5.3.2　类

1．类的基础知识

说到类，就不得不提面向过程编程和面向对象编程。下面用两个例子说明这两种编程方式在编码上的区别，如图 5-3-9 和图 5-3-10 所示。

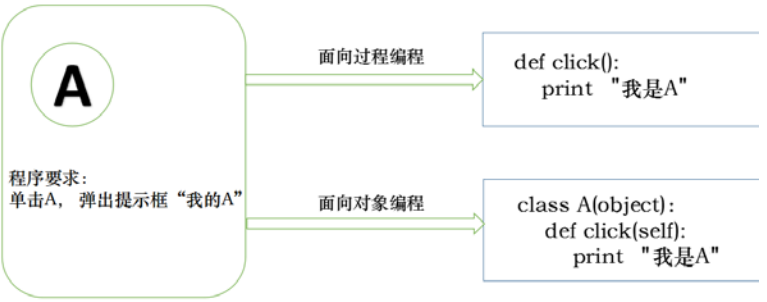


图 5-3-9　案例 1

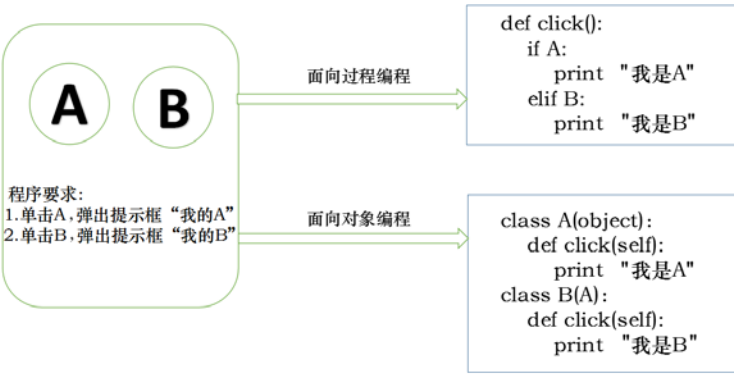


图 5-3-10　案例 2

（1）在面向对象编程中最重要的概念是类（Class）和实例（Instance）。

（2）类是抽象的模板，可以将其理解为一个基础模板，比如 Student 类。而实例是根据类创建出来的一个具体对象。

“物以类聚，人以群分”，可以从这句话来理解编程中的类。类应该是有共同特点的事物的集合，比如人类、动物类。类中的对象应该有共同的特点，而这些特点对应编程中的类的共同属性和方法。而具体的对象之间又或多或少有差别，比如在“鸟”这个类中，每只鸟又有自己的特点。

（3）每个对象都拥有相同的方法，但各自的数据可能不同。

每个对象都拥有相同的方法，实例拥有类中所有的方法；无论实例的名称如何不同，它们的方法都是一样的。比如，类是 Student，那么不管是实例 lilei，还是实例 hanmeimei，只要它们都是 Student 的实例，它们所拥有的方法都是相同的。

每个实例的数据可能不同，这是因为在使用实例时，如果需要在这个类中传入参数作为初始数据，那么每个实例可能会被传入不同的（或者相同的）数据。比如数据库类，如果传入的是不同的数据库连接串，则实例的数据就不同了。

2．类的使用

（1）在 Python 中，定义类是通过 class 关键字来实现的：

```
class Student(object):
    Pass
```

- class 后面紧接着是类名，即 Student。类名通常采用单词首字母大写的形式命名，比如 StudentCore。
- (object)表示该类是从哪个类继承来的。如果没有合适的继承类，则通常使用 object 类，这是所有类都会继承的类。

（2）实例化一个类是通过在类名后面加小括号“()”来实现的：

```
pupil= Student()
```

- “小学生”是“学生”这个类的一个实例。所谓实例，可以将其理解为类的化身。类不能被直接使用，只能先将其实例化，然后用实例代表类，进而调用类中的方法处理数据。那么实例是怎么代表类的呢？在定义类及其方法时，有一个参数叫 self，它就是串联实例与类之间的关键参数，可以简单地理解为“self=实例名”。
- 类的下面有属性和方法。可以将属性理解为静态的数据，将方法理解为动态的处理函数。比如，在下面的代码中定义了“学生”类，其中_ _init_ _是属性，print_score 是方法。

```
# -*- coding:utf-8 -*-
class Student(object):
```

```
    def _ _init_ _(self, name, score):
        self.name = name
        self.score = score
    def print_score(self):
        print ('%s: %s' % (self._ _name, self._ _score))
```

（3）实例化上面的类，然后分析 Python 如何利用实例化类调用类中的方法。

完整的实例化类与调用过程如下：

```
pupil= Student(name='lilei',score=99)
pupil.print_score()
```

代码解释如下：

按照前面的说法，self=实例名，那么 pupil= Student(name='lilei',score=99)，实例化类的第一步是执行_ _init_ _语句，即 Student(self='pupil',name='lilei',score=99)，则可知 pupil.name ='lilei'，pupil.score=9。实例化类的第二步是执行 pupil.print_score()语句，调用实例的方法就是调用类的方法，而这个方法是按照格式打印出 pupil.name:pupil.score，即 lilei：99。

3．类的优势

（1）类的访问限制。

所谓类的访问限制，即在类的外部不能直接调用类中的数据。在 Python 中，在变量前加“_ _”可将变量置为私有变量，使其只能在类中被使用，不能被类之外的其他函数（方法）调用。下面代码中的_ _name 参数表示的就是私有变量。

```
# -*- coding: utf-8 -*-
#_ _author_ _ = '大婶 N72'
class Student(object):
    def _ _init_ _(self, name, score):
        self._ _name = name
        self.score = score
    def print_score(self):
        print ('%s: %s' % (self._ _name, self._ _score))
MeiMei=Student('hello',99)
```

```
print(MeiMei.score)
print(MeiMei._ _name)
```

代码的运行结果很好地佐证了加了“_ _”的参数是不能被调用的，没加“_ _”的参数是可以直接被调用的，如图 5-4-11 所示。

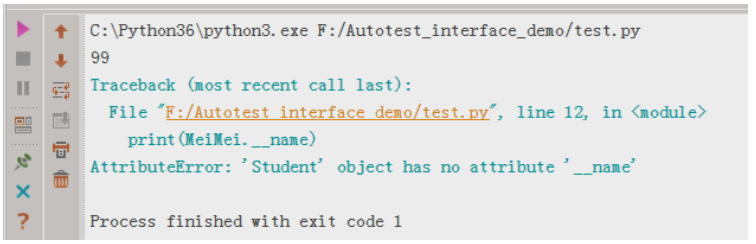


图 5-3-11　代码的运行结果

（2）类的继承。

所谓“继承”，从字面上看很好理解：如果一个类继承了另外一个类，则这个类就有了被继承的类的所有方法和属性。我们可以直接使用这些方法和属性，无须再单独写。下面举一个例子来解释什么是继承。类的基本定义如下：

```
# -*- coding: utf-8 -*-
#_ _author_ _ = '大婶 N72'
#父类
class Parent(object):
    def print_self(self):
        return "我是父类"
#子类继承父类
class Student(Parent):
    def _ _init_ _(self, name, score):
        self._ _name = name
        self.score = score
    def print_score(self):
        print ('%s: %s' % (self._ _name, self._ _score))
MeiMei=Student('hello',99)
print(MeiMei.score)
#直接在子类中使用父类的方法
print(MeiMei.print_self()
```

在上面的例子中，子类 Student 继承了父类 Parent。方法是在类名称后面的小括号中加上父类的名称。子类继承了父类，则有了父类 Parent 的方法 print_self，从而可以直接在自己的实例名中使用。

（3）类的多态。

如果子类继承了父类，但是又想改变父类的方法，则这时无须修改父类的方法，只需要在子类中添加相同的方法名就可以起到覆盖的作用，这就是多态。读者运行下面的代码可以了解什么是多态。

```
# -*- coding: utf-8 -*-
#_ _author_ _ = '大婶 N72'
#父类
class Parent(object):
    def print_self(self):
        return "我是父类"
#子类继承父类
class Student(Parent):
    def _ _init_ _(self, name, score):
        self._ _name = name
        self.score = score
    def print_score(self):
        print ('%s: %s' % (self._ _name, self._ _score))
    def print_self(self):
        return "我是子类"
MeiMei=Student('hello',99)
print(MeiMei.score)
#直接在子类中使用父类的方法
print(MeiMei.print_self())
```

5.3.3 条件判断

Python 程序在运行后之所以能够得到不同的结果，是因为根据不同的条件执行了不同的语句。条件判断是 Python 程序运行的基本逻辑之一。比如，根据不同的分数显示不同的等级，在 Python 程序中可用下列 if 语句来实现：

```
# -*- coding:utf-8 -*-
#_ _author_ _ = '大婶 N72'
score=90
if score>=90:
    print("A")
```

在上面的代码中，如果 score 大于或等于 90，则使用 print 函数输出“A”这个字符。注意，if 语句后面是冒号，在其条件为“真”时，才会执行冒号后面的语句，所以上面的代码的输出结果是“A”。当然，完整的 if 语句肯定不会这么简单，例如下面的代码增加了一些判断条件：

```
# -*- coding:utf-8 -*-
#_ _author_ _ = '大婶 N72'
score=60
if score>=90:
    print("A")
else:
    print("非 A")
```

在上面的代码中，增加了 else 语句及其后面要执行的语句。else 的意思是“否则”，即不满足其前面所有的 if 条件时就执行 else 后面的语句，而 else 后面是没有其他限制条件的，上面代码的输出结果是“非 A”。下面通过 elif 语句再增加一些判断条件。

```
# -*- coding:utf-8 -*-
#_ _author_ _ = '大婶 N72'
score=87
if score==100:
    print("满分")
elif 90<=score<100:
    print("A")
elif 70<=score<90:
    print("B")
elif 60<=score<70:
    print("C")
elif score>100 or score<0:
    print("别扯了")
else:
    print("不及格")
```

elif 其实是 else…if，即“否则……如果”，增加 elif 就是再增加一种判断情况。elif 语句是允许有多个条件存在的。在不出现异常的情况下，程序会按照顺序从上往下判断，直到 else。在这个过程中，如果哪个判断条件成立，则执行该条件下的语句。

总结一下， if 条件判断语句的语法机制是：

```
# -*- coding:utf-8 -*-
#_ _author_ _ = '大婶 N72'
if <条件判断 1>:
    <执行 1>
elif <条件判断 2>:
    <执行 2>
elif <条件判断 3>:
    <执行 3>
else:
    <执行 4>
```

备注：这里提到条件判断语句成立（即为 True），那么在 Python 3 中哪些情况是 True 呢？除日常的真假判断外，在 Python 3 中还有一些特殊的数据类型能代表 True 和 False，但是不建议用特殊的符号来判断，因为其易读性不好。下面是判断真假的几种情况：

```
# -*- coding:utf-8 -*-
#_ _author_ _ = '大婶 N72'
#假：[],{},0,0.0,None,False,""
#真：1,"TEST"
if "":
    print("真")
else:
    print("假")
```

5.3.4 异常处理

简单的异常处理主要依靠内置异常处理结构体和 logging 日志，代码结构如下：

```
import logging
try:
    ...
except ValueError as e:#异常判断，出现 ValueError 错误时处理机制
```

```
    ...
except ZeroDivisionError as e:#异常判断，出现 ZeroDivisionError 错误时处理机制
    logging.exception(e)
finally:
    ...
```

下面的例子解释了“内置异常处理结构体+logging 日志”是如何捕获异常情况的：

```
# -*- coding:utf-8 -*-
import logging
class Test(object):
    def _ _init_ _(self,num):
        self.num=num
    def unnormal(self):
        try:
            print('开始测试异常情况')
            r = 10 / self.num
            print 'result:', r
        except TypeError as e:
            print 'TypeError:' as e
        except ZeroDivisionError as e:
            logging.exception(e)
        finally:
            print 'finally finish!'
```

读者可以分别使用以下的实例化传参来测试异常处理机制：

```
op_test1=Test(num=2)
op_test1.unnormal()
op_test2=Test(num=test)
op_test2.unnormal()
op_test3=Test(num=0)
op_test3.unnormal()
```

5.3.5 Python 3 代码注释

注释就是一段解释，可以起到备注的作用。在团队合作中，每个人编写的代码经常会被多人调用，为了让别人能更容易理解代码，非常有必要使用注释。在每个 py 文件的

开头加上代码功能的注释，这样，其他人看到注释就大概知道该 py 文件所实现的功能和调用方式了。所以，读者需要养成注释的习惯并使用规范的注释方式。

在 Python 中，代码的注释方式有两种——单行注释和多行注释。

- 单行注释：“#”符号常被用来对单行文字或代码做注释。使用“#”时，它右边的任何数据都被当作是注释。比如：

```
# -*- coding:utf-8 -*-
#_ _author_ _ = '大婶 N72'
# 这是一个注释
print("Hello, World!")#输出 hello world
```

- 多行注释：用三个单引号（'''）或者三个双引号（"""）将注释内容括起来，也被称为块注释。比如：

```
# -*- coding:utf-8 -*-
#_ _author_ _ = '大婶 N72'
'''
print("Hello, World!")
print("I'm fine!")
print("thank you !")
'''

"""
print("Hello, World!")
print("I'm fine!")
print("thank you !")
"""
```

在 PyCharm 工具中，可以使用快捷键很方便地添加注释和取消注释（快捷键是“Ctrl+?”）。将鼠标光标移动到需要加注释的语句上，按“Ctrl+?”键即可添加或删除“#”。而对于多行注释，在输入三个单引号“''”或者三个双引号“"""”时，会自动关联出另一对三个单引号“''”或者三个双引号“"""”。

5.4　PyCharm 使用基础

工欲善其事，必先利其器。好的开发工具，会令代码编写成为一件轻松而愉快的事情，能显著提高编程效率。这里向读者介绍两款工具，一款是重量级工具，另一款是轻量级工具。下面分别是这两款工具的代码风格，图 5-4-1 所示的是 PyCharm，图 5-4-2 所示的是 Sublime Text 3。

图 5-4-1　PyCharm

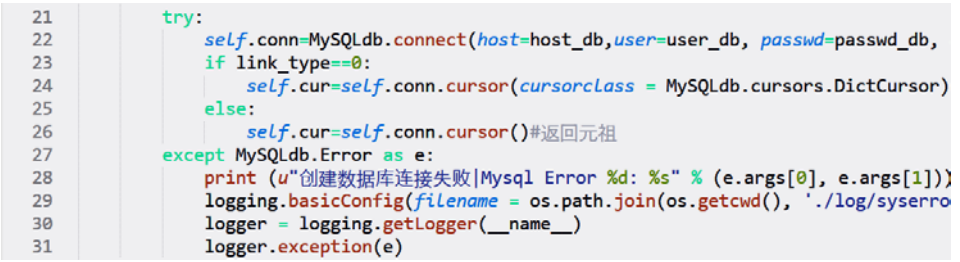


图 5-4-2　Sublime Text 3

5.4.1　为什么选择 PyCharm

PyCharm 有以下优点：

（1）PyCharm 拥有一般 IDE 所具备的功能，比如，调试、语法高亮、项目管理、代码跳转、智能提示、自动完成、单元测试、版本控制……

（2）PyCharm 还提供了一些很好的功能，可用于 Django 开发及其他 Web 开发框架。

（3）对初学者而言，PyCharm 比较完整，使用成本低，不需要安装额外的插件。

（4）对有基础者而言，PyCharm 与其他工具类似，切换成本低。

PyCharm 的缺点是：占用资源多。

5.4.2 PyCharm 使用基础

1. 新建工程、文件

该步骤已经在 5.2.2 小节详细介绍过，这里不再赘述。

2. 代码运行

运行指定的 py 代码文件，在代码名称或者代码中的任何地方单击鼠标右键，从弹出的快捷菜单中选择“Run×××”命令，如图 5-4-3 所示。

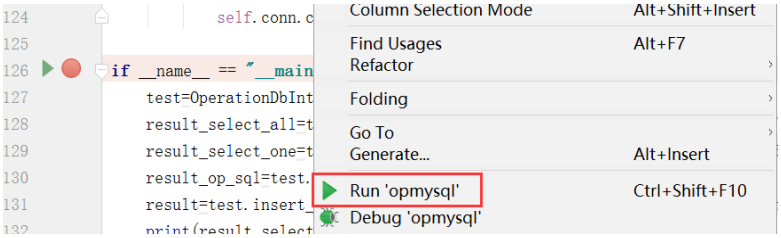


图 5-4-3 在 PyCharm 中运行代码文件

在 PyCharm 中会显示输出信息，如图 5-4-4 所示。

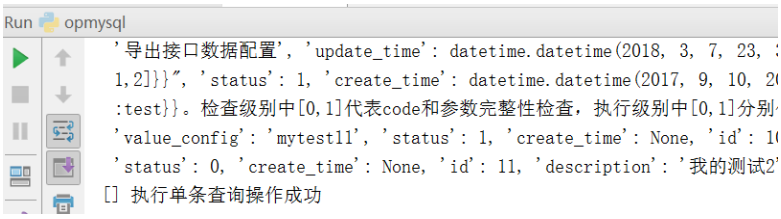


图 5-4-4 显示输出信息

3. 调试代码

调试代码（俗称“打断点”）是程序开发中重要的一环，可以帮助开发者定位和查

找问题，是开发者必须要会的技能之一。在 PyCharm 中可按照以下步骤调试代码。

（1）在想要调试的代码区域前/后单击鼠标，即“打断点”，如图 5-4-5 所示。

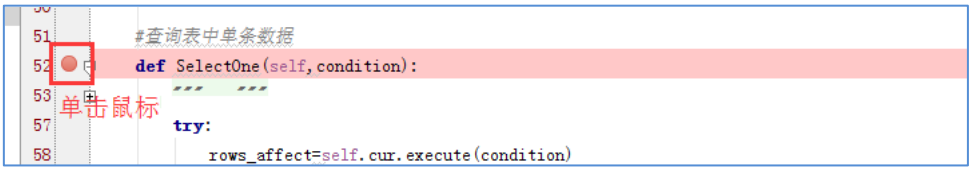


图 5-4-5　“打断点”

（2）开启调试模式，即在代码中的任意位置单击鼠标右键，在弹出的快捷菜单中选择“Debug×××”命令，如图 5-4-6 所示。

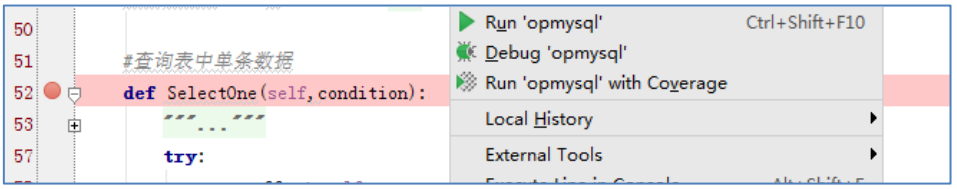


图 5-4-6　开启调式模式

（3）按 F8 键开始调试代码。当代码运行到开始调试的位置时，每按一次 F8 键就执行一步。在这个过程中，可以通过下方的 Debugger 窗口查看过程数据，如图 5-4-7 所示。

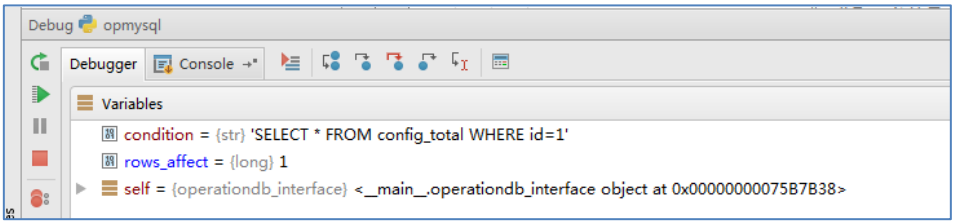


图 5-4-7　查看过程数据

（4）在调试过程中，系统会用蓝色底色标志出下一步要执行的代码，提示调试者即将走到哪一步。而 Debugger 窗口中的蓝色文字，则表示当前代码执行的结果。例如，图 5-4-8 中所示的第 59 行代码是将要执行的，而 Debugger 窗口中蓝色字体的 rows_affect 则是已经执行的第 58 行代码的结果。

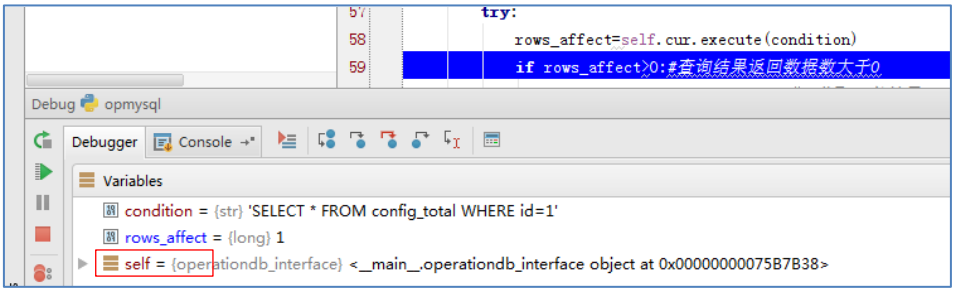


图 5-4-8　查看进度

（5）在中断调试后，需要关闭调试模式，否则会开启很多调试窗口，影响其他调试过程的判断。关闭的方法如图 5-4-9 所示。

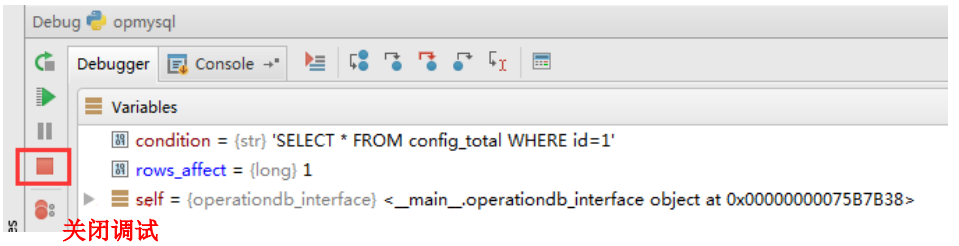


图 5-4-9　关闭调试模式

4．代码跳转

代码跳转可以帮助我们查看所调用的方法或者函数本身的代码。设置代码跳转的方法是：在按住 Ctrl 键的同时单击该方法或函数。例如，在 opmysql.py 中使用了 PyMySQL，如图 5-4-10 所示。

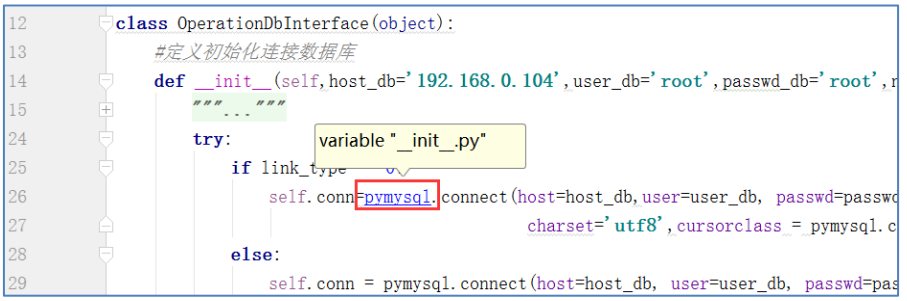


图 5-4-10　代码跳转

5.5 补充知识点

5.5.1 MySQLdb 与 PyMySQL

MySQLdb 与 PyMySQL 都是在 Python 中操作 MySQL 数据库的包，只不过二者所支持的 Python 版本不同。MySQLdb 支持 Python 2，PyMySQL 支持 Python 3。这是读者需要注意的地方。

5.5.2 Python 命名规则

（1）模块或包全部使用小写字母的命名方式，并且以下画线分隔单词，如：get_sign。

（2）类或异常使用每个单词首字母大写的命名方式，如：OperationDbInterface。

（3）全局或类常量全部使用大写字母的命名方式，并且以下画线分隔单词。

（4）其余变量（包括方法名、函数名和普通变量名）则是全部使用小写字母的命名方式，并且以下画线分隔单词，如：op_sql。

（5）以上的内容如果是 Python 内部的，则使用双下画线开头命名，如：_ _init_ _，_ _del_ _。

5.5.3 self.cur.scroll 源码分析

如果在 select_all 方法中用到了 self.cur.scroll(0, mode='absolute')，那么在运行代码时要将游标的索引加到初始位置，具体介绍如下：

```
self.cur.execute(condition)            #在游标下执行 SQL 语句
self.cur.scroll(0,mode='absolute')     #让游标的索引回到初始位置
results =self.cur.fetchall()           #返回游标中的所有结果
```

假如测试数据库表中的数据有 3 条，则表结构如图 5-5-1 所示。

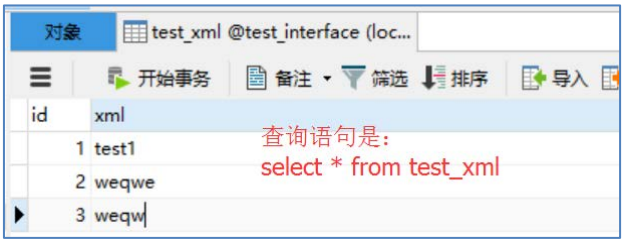


图 5-5-1　表结构与数据查询

（1）在游标下执行 SQL 语句，则执行的结果被先放在游标中。

（2）在游标下执行 scroll 方法，如果不清楚这一步具体是做什么的，则可以先看 Python 源码是怎么解释的，如图 5-5-2 所示。

```
def scroll(self, value, mode='relative'):
    """Scroll the cursor in the result set to a new position according
    to mode.

    If mode is 'relative' (default), value is taken as offset to
    the current position in the result set, if set to 'absolute',
    value states an absolute target position."""
    self._check_executed()
    if mode == 'relative':
        r = self.rownumber + value
    elif mode == 'absolute':
        r = value
    else:
        self.errorhandler(self, ProgrammingError,
                          "unknown scroll mode %s" % repr(mode))
    if r < 0 or r >= len(self._rows):
        self.errorhandler(self, IndexError, "out of range")
    self.rownumber = r
```

r=0

self.rownumber=0，即序列=0

图 5-5-2　scroll 方法的源码

（3）在游标下执行 fetchall 方法，看一下 Python 源码是怎么解释的，如图 5-5-3 所示。

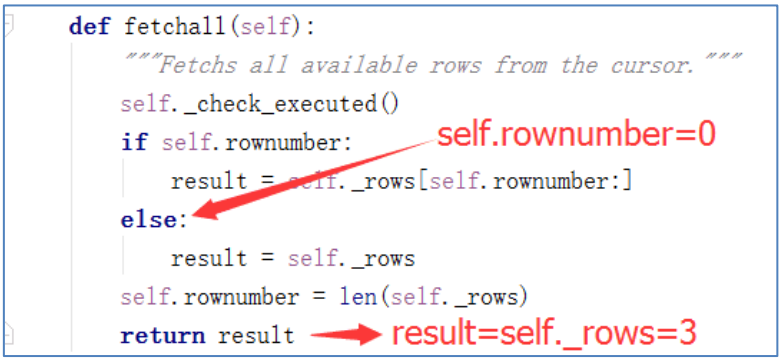


图 5-5-3　fetchall 方法的源码

（4）设置游标的位置。

可以通过 cursor.scroll(position, mode="relative | absolute")方法，来设置相对位置游标和绝对位置游标。

方法参数描述：

① position：游标位置（游标位置从 0 开始）。

② mode：游标位置的模式，包括以下两种。

- relative：默认模式，相对当前位置（即执行 scroll 方法时游标的位置）。
- absolute：绝对位置。

例如：

mode=relative, position=1：设置游标位置为“当前位置+1”，即向下移动一个位置。

mode=absolute, position=2：将游标移动到索引为 2 的位置，无论当前位置在哪里。

5.5.4　主流数据库的分类

在当今的互联网中，最常见的数据库模型主要有两种——关系型数据库和非关系型数据库，如图 5-5-4 所示。

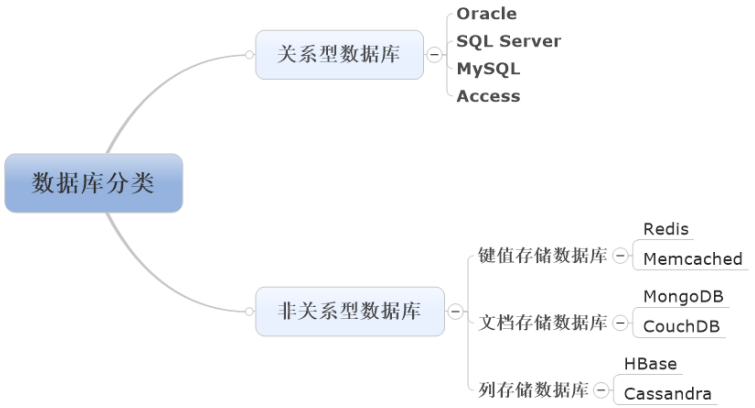


图 5-5-4　数据库的分类

1．关系型数据库

关系型数据库是把一些复杂的数据结构归结为简单的二元关系（即二维表格形式），其对数据的操作几乎全部建立在一个或多个关系表格中，通过对这些关系表格的分类、合并、连接或选取等运算来实现数据管理。所以，当数据很大、很复杂时，关系型数据库的执行效率会下降得厉害。

2．非关系型数据库

非关系型数据库从字面上就能理解，数据的存储不是通过关键字建立关系的。其实它是简洁版的关系型数据库，其在高并发、数据量大的情况下效率更高。

非关系型数据库分为以下几种。

（1）键–值（Key-Value）存储数据库。

键值存储数据库通过键来添加、查询或者删除数据。因为使用主键访问，所以会具有很高的性能及扩展性。

（2）列存储（Column-Oriented）数据库。

列存储数据库将数据存储在列族中，一个列族中存储着经常被一起查询的相关数据。

比如对于人，经常被一起查询的是姓名和年龄，而不是薪资。在这种情况下，“姓名”和“年龄”会被放到一个列族中，“薪资”会被放到另一个列族中。这种数据库通常被用来分布式存储海量数据。

（3）面向文档（Document-Oriented）数据库。

面向文档数据库将数据以文档的形式存储。每个文档都是自包含的数据单元，是一系列数据项的集合。每个数据项都有一个名词与对应值。其值既可以是简单的数据类型（如字符串、数字和日期等），也可以是复杂的数据类型（如有序列表和关联对象）。数据存储的最小单位是文档。同一个表中存储的文档属性可以是不同的，可以使用 XML、JSON 或 JSONB 等多种形式存储数据。

5.5.5　MySQL 的基本语法

在本书中使用最多的就是增加、删除、修改、查询这 4 种操作，下面介绍在 MySQL 中如何实现这几种基本操作。

（1）增加数据：insert into *table1*(*field*1,*field*2) values(*value*1, *value*2)

（2）删除数据：delete from *table*1 where *范围*

（3）修改数据：update *table*1 set *field*1=value1 *where* *范围*

（4）查询数据：select * from *table*1 where *范围*

（5）嵌套查询：select * from *table*1 where *id* in (select *id* from *table*2)

（6）清空表数据：TRUNCATE TABLE *table*1

下面举例具体说明。

1．插入单条数据

```
 INSERT INTO 'config_total' ('key_config', 'value_config', 'description',
'status') VALUES ('test', 'value_test', '测试配置', '1');
```

解释：插入 config_total 表中的 key_config、value_config、description、status 字段，

值是 test、value_test、测试配置和 1。注意，表字段与值要一一对应。

2．插入多条数据

```
INSERT INTO 'config_total' ('key_config', 'value_config', 'description', 'status') VALUES ('test', 'value_test', '测试配置 1', '1'),('test', 'value_test', '测试配置 2', '1');
```

3．结合 Python 以参数化插入单条数据

```
INSERT INTO 'config_total' ('key_config', 'value_config', 'description', 'status') VALUES(%s,%s,%s,%s,) %(param1,param2,param3,param4);
```

4．删除数据（一般不用，都是逻辑删除，即修改状态）

```
DELETE FROM 'config_total' WHERE 'key_config'='test' AND 'status'='1';
```

解释：从 config_total 表中删除数据，条件是'key_config'='test'和'status'='1'同时满足。也可以使用 OR，即只要满足一个条件。

5．修改数据

```
UPDATE 'config_total' SET 'value_config'='config1' WHERE 'key_config'='test' AND 'status'='1';
```

解释：更新 config_total 表，设置 value_config 字段值为 config1，条件是'key_config'='test'和'status'='1'同时满足。也可以使用 OR，即只要满足一个条件。

6．结合 Python 参数化修改数据

```
"UPDATE config_total SET key_config='%s',value_config='%s',value_config='%s' WHERE id=%s" %(param1,param2,param3,param4)
```

7．查询所有数据

```
SELECT * FROM 'config_total';
```

8．查询符合指定条件数据的所有字段值

```
SELECT * FROM 'config_total' WHERE 'key_config'='test';
```

9．查询符合指定条件数据的指定字段值

```
SELECT 'key_config', 'value_config' FROM 'config_total' WHERE 'key_config'='
```

```
test';
```

10．结合 Python 参数化查询数据

```
 "SELECT * from case_interface where id_task=%s and case_status=1" %id_task
```

11．结合 Python 参数化嵌套查询数据

```
 "SELECT id_module,desc_case FROM case_module where id_module in (SELECT id
from module where status_module=1 and name_module='%s')" %name_module
```

6

用 Python 发送 HTTP 请求

本章讲什么：

1. 用 Python 如何发送 HTTP 请求；
2. 本章涉及的 Python 部分语法。

本章是 Python 接口自动化测试的核心内容之一。因为只有成功发送 HTTP 请求，才能进行后续的数据处理和结果校验。

由于测试数据存储在数据库中，本章会从数据库表中获取接口、测试数据，并把测试结果写入数据库表中。这是第 5 章所介绍的基本操作方法，所以，在开始本章学习之前读者应熟练使用 Python 操作 MySQL 数据库完成增加、删除、修改、查找操作。

本章是从“数据库接口测试用例表”中获取指定数据，然后使用 requests 模块发送 HTTP 请求，并获取返回包数据。

6.1 准备工作

准备工作是初始化测试用例表及其数据。

这里通过 SQL 语句添加测试数据，以后可以尝试着开发 Web 页面去添加测试数据。

1．新建数据库接口用例表，并初始化测试数据

具体代码如下：

```
CREATE TABLE 'case_interface' (
  'id' int(2) NOT NULL AUTO_INCREMENT,
  'name_interface' varchar(128) NOT NULL COMMENT '接口名称',
  'exe_level' int(2) DEFAULT NULL COMMENT '执行优先级，0 代表 BVT',
  'exe_mode' varchar(4) DEFAULT NULL COMMENT '执行方式：POST、GET，默认是 POST
方式',
  'url_interface` varchar(128) DEFAULT NULL COMMENT '接口地址:直接使用 HTTP
开头的详细地址',
  'header_interface' text COMMENT '接口请求的头文件，有则使用，无则不用',
  'params_interface' varchar(256) DEFAULT NULL COMMENT '接口请求的参数',
  'result_interface' text COMMENT '接口返回结果',
  'code_to_compare' varchar(16) DEFAULT NULL COMMENT '待比较的 Code 值，用户自
定义比较值，例如 ReturnCode 和 Code 等，默认 ReturnCode',
  'code_actual' varchar(16) DEFAULT NULL COMMENT '接口实际 Code 返回值',
  'code_expect' varchar(16) DEFAULT NULL COMMENT '接口预期 Code 返回值',
```

```
  'result_code_compare` int(2) DEFAULT NULL COMMENT 'Code 比较结果，1-pass，
0-fail，2-无待比较参数，3-比较出错，4-返回包不合法，9-系统异常',
  'params_to_compare' varchar(256) DEFAULT NULL COMMENT '接口比较参数集合，用
于比较参数的完整性',
  'params_actual' text COMMENT '接口实际返回参数',
  'result_params_compare' int(2) DEFAULT NULL COMMENT '参数完整性比较结果，
1-pass，0-fail，2-获取参数集错误，9-系统异常',
  'case_status' int(2) DEFAULT '0' COMMENT '用例状态，1-有效，0-无效',
 'create_time' timestamp NULL DEFAULT CURRENT_TIMESTAMP COMMENT '创建时间',
  'update_time'  timestamp  NULL  DEFAULT  CURRENT_TIMESTAMP  ON  UPDATE
CURRENT_TIMESTAMP COMMENT '更新时间',
  PRIMARY KEY ('id')
 ) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8 COMMENT='接口用例表';

 INSERT INTO 'case_interface' ('name_interface', 'exe_level', 'exe_mode',
'url_interface', 'header_interface', 'params_interface', 'code_to_compare',
'code_expect',    'params_to_compare',    'case_status',    'create_time',
'update_time')       VALUES       ('getIpInfo.php',       0,       'GET',
'http://ip.taobao.com/service/getIpInfo.php',
'{\'Host\':\'ip.taobao.com\'}',       'ip=63.223.108.4','code',       '0',
'[\'code\',\'data\',\'country\']',1,NOW(),NOW());
```

至此完成了准备工作。下面开始代码编写和调试。

2．对 case_interface 表中的一些字段追加说明

下面是对 case_interface 表中的一些字段追加说明。如果读者不感兴趣，则可以先忽略，等需要了解该字段的意义及如何赋值时再看。

- code_to_compare：待比较的关键参数的名称。由于每个接口要检查的关键参数名称可能都不一样，所以需要告知代码把哪个参数作为关键参数。比如，有的接口返回的参数是 returncode，有的返回的参数是 Code。如果要比较这些参数的值，则需要在接口表中指定。
- params_to_compare：参数完整性。有时需要测试接口是否返回了多个参数，这时只需要检查这几个参数是否存在。比如，对于一个接口，前端页面需要提供 3 个参数，而实际只提供了两个参数，或者实际提供的 3 个参数并不是接口协议中约

定的，那前端页面在解析时就会出错。该字段用于告诉代码，要检查预期的参数是否都存在。

补充：

还有一点需要检查，即返回包的结构。比如，返回包的结构按照协议应该是下面的结构体，这里也不关心参数值。这里与前面参数完整性检查的区别是，这里更看重结构和顺序。这个知识点读者可以自行拓展，不在本书的介绍范围。

```
{
    "code": 0000,
    "message": "success",
    data: [
        {
            "id": 1,
            "phone": 15156070000,
            "type": "mobile"
        }
    ]
}
```

6.2　发送 HTTP 请求实例

6.2.1　用 Python 发送 HTTP 请求的流程

图 6-2-1 是用 Python 代码发送 HTTP 请求的流程。

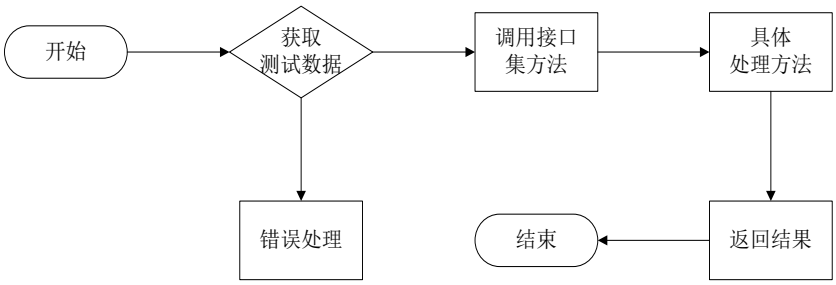


图 6-2-1　用 Python 发送 HTTP 请求的流程

流程图的说明如下。

（1）开始：直接运行 reuqest.py，或者用主方法调用 reuqest.py 中的方法。reuqest.py 就是后面要建的处理 HTTP 请求的代码。

（2）获取测试数据：用第 5 章中介绍的数据库操作测试用例表中的数据。当然，这一步肯定涉及逻辑的判断：如果获取的测试数据正确，则继续后面的调用接口集方法；而如果获取测试数据失败或异常，则会做一次错误处理，而不是直接抛出异常信息。

（3）调用接口集方法：按接口请求数据的类型去调用不同的处理方法。接口集方法支持被外部代码所调用，但其实际的底层代码是不支持直接被调用的，这达到了底层代码封装的效果。

（4）具体处理方法：在处理完 HTTP 请求后，返回结果包，则该层级的工作完成。

图 6-2-2 所示是用 Python 发送 HTTP 请求流程中第（3）步（调用接口集方法）的细化。

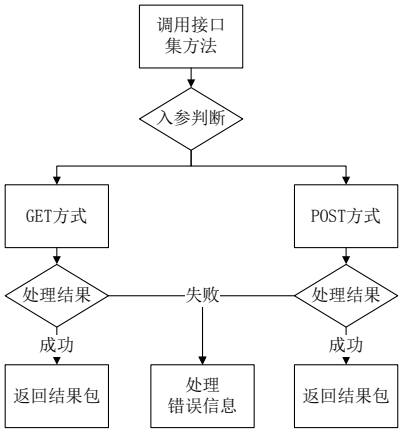


图 6-2-2　调用接口集方法的细化

对于该过程的解释如下：

（1）依据接口集方法 http_request 中的特定参数，来判断具体要调用的内部方法。

（2）获取到指定的参数值后，http_request 会调用指定的内部方法去处理 HTTP 请

求，并获得处理结果。而内部方法会对处理结果进行一层封装后将其返回给 http_request，http_request 得到返回数据后再返给上一层调用方。

6.2.2　用 Python 操作 HTTP 请求的代码

1．新建 request.py 文件

在 common 目录下新建一个 request.py 文件。可以按照下面的方法新建：

（1）右击“common”文件夹，在弹出的菜单中选择“New”→“Python file”命令，新建.py 文件，将“Name”设为“request”，如图 6-2-3 所示。

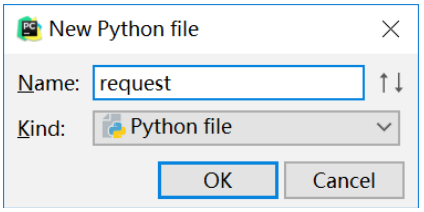


图 6-2-3　新建 request.py 文件

（2）单击“OK”按钮后，新建工程及其目录下的.py 文件成功，目录结构如图 6-2-4 所示。

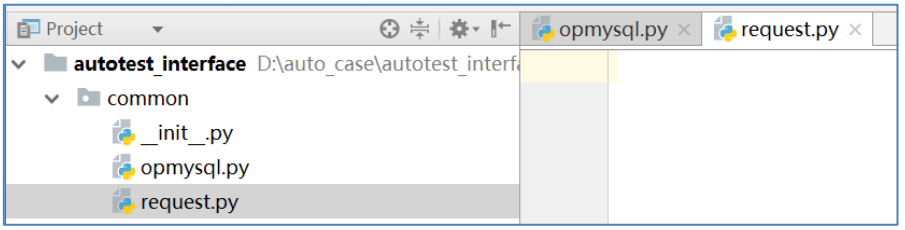


图 6-2-4　新建 request.py 文件成功

2．在.py 文件中创建代码

代码：request.py

```
1 # -*- coding:utf-8 -*-
2 #_ _author_ _ = '大婶 N72'
3 '''
```

```
4  封装 HTTP 请求操作
5  1.http_request 是主方法，直接供外部调用
6  2._ _http_get、_ _http_post 是实际底层分类调用的方法
7  '''
8  import requests,os,logging
9  from common import opmysql
10  from public import config
11  class RequestInterface(object):
12      #定义处理不同类型的请求参数，包含字典、字符串、空值
13      def _ _new_param(self,param):...
……
27      # POST 请求，参数在 body 中
28      def _ _http_post(self,interface_url,headerdata,interface_param):
……
54      # GET 请求，参数在接口地址后面
55      def _ _http_get(self,interface_url,headerdata,interface_param):
……
83      # 统一处理 HTTP 请求
84      def http_request(self,interface_url,headerdata,interface_param,
request_type):
……
108  if _ _name_ _ == "_ _main_ _":
109      test_interface=RequestInterface()#实例化 HTTP 请求类
110
test_db=opmysql.OperationDbInterface(host_db='192.168.0.104',user_db='roo
t',passwd_db='root',name_db='test_interface',port_db=3306,link_type=0)#实
例化 MySQL 处理类
111      sen_sql="select exe_mode,url_interface,header_interface,
params_interface from case_interface where name_interface='getIpInfo.php'
and  id=1"
112      params_interface=test_db.select_one(sen_sql)
113      if params_interface['code']=='0000':
114          url_interface=params_interface['data']['url_interface']
115          temp=params_interface['data']['header_interface']
116          headerdata=eval(params_interface['data']['header_interface'])
#将 unicode 转换成字典
117          param_interface=params_interface['data']['params_interface']
118          type_interface=params_interface['data']['exe_mode']
119          if url_interface!='' and headerdata!='' and param_interface!=''
```

```
and type_interface!='':
 120            result=test_interface.http_request(interface_url=
url_interface,headerdata=headerdata,interface_param=param_interface,reque
st_type=type_interface)
 121            if result['code']=='0000':
 122                result_resp=result['data']
 123                test_db.op_sql("UPDATE case_interface SET
result_interface='%s' WHERE id=1" %result_resp)#将结果更新到 case_interface
表中
 124                print("处理 HTTP 请求成功，返回数据是：%s" %result_resp)
 125            else:
 126                print("处理 HTTP 请求失败")
 127        else:
 128            print("测试用例数据中有空值")
 129     else:
 130         print("获取接口测试用例数据失败")
```

上面是 request.py 文件的一部分，大致能看出这个文件的结构。具体每个方法是怎么实现的，会在下面逐步进行讲解。这里先把握整体的结构。

该代码是一个封装的、用 Python 发送 HTTP 请求的类。类名是 RequestInterface，类的下面有 4 个方法。

- _ _init_ _：初始化数据。
- http_request：统一处理 HTTP 请求，也是可被直接调用的方法。
- _ _http_post：处理 POST 类型接口请求的内部方法。
- _ _http_get：处理 GET 类型接口请求的内部方法。

大家也可以根据实际的需要新增或合并对应的方法。总体的设计思想是，将常用的方法预先写好，需要时直接调用。

最后的 if _ _name_ _ == "_ _main_ _"中写入的是，实例化类和调用类中的方法。

下面对该.py 文件进行详细介绍。

代码：request.py

```
1 # -*- coding:utf-8 -*-
```

```
2  #_ _author_ _ = '大婶 N72'
3  '''
4  封装对 HTTP 请求操作
5  1.http_request 是主方法，直接供外部调用
6  2._ _http_get、_ _http_post 是实际底层分类调用的方法
7  '''
8  import requests,os,logging
9  from common import opmysql
10  from public import config
11  class RequestInterface(object):
```

下面是对上一段代码的解释。

- 第 8 行代码：使用第三方包 requests 发送 HTTP 请求，还需要使用数据库操作模块 opmysql，该模块需要从 common 目录下导入，导入时使用“from common import 模块名”即可。
- 第 11 行代码：定义一个类，其名称为 RequestInterface，其继承的父类默认使用 object 超类（前提是该类没有父类）。

代码：request.py

```
12      #定义处理不同类型的请求参数，包含字典、字符串、空值
13      def _ _new_param(self,param):
14          try:
15              if isinstance(param,str) and param.startswith('{'):
16                  new_param=eval(param)
17              elif param==None:
18                  new_param=''
19              else:
20                  new_param=param
21          except Exception as error:#记录日志到 log.txt 文件
22              new_param=''
23              logging.basicConfig(filename = config.src_path +
'/log/syserror.log',level = logging.DEBUG,format='%(asctime)s %(filename)
s[line:%(lineno)d] %(levelname)s %(message)s')
24              logger = logging.getLogger(_ _name_ _)
25              logger.exception(error)
26          return new_param
```

下面是对上一段代码的解释。

第 12~26 行代码：对特殊情况的接口请求参数做处理，返回处理后的接口请求参数。

- 第 15、16 行代码：进行参数类型判断。如果接口请求参数是一个字符串类型的字典，则用 eval 函数将其还原成字典形式。
- 第 17、18 行代码：如果处理请求参数是 None，则直接返回一个空字符串。
- 第 19、20 行代码：如是其他情况，则都返回本来的参数。

代码：request.py

```
27     # POST 请求，参数在 body 中
28     def _ _http_post(self,interface_url,headerdata,interface_param):
29         '''
30         :param interface_url: 接口地址
31         :param headerdata: 请求头文件
32         :param interface_param: 接口请求参数
33         :return:字典形式结果
34         '''
35         try:
36             if interface_url!='':
37                 temp_interface_param = self._ _new_param(interface_param)
38                 response = requests.post(url=interface_url,
headers=headerdata,data=temp_interface_param,verify=False,timeout=10)
39                 if response.status_code==200:
40                     durtime = (response.elapsed.microseconds) / 1000  # 发
起请求和响应到达的时间，单位 ms
41                     result={'code':'0000','message':'成功
','data':response.text}
42                 else:
43                     result={'code':'2004','message':'接口返回状态错误
','data':[]}
44             elif interface_url == '':
45                 result={'code':'2002','message':'接口地址参数为空
','data':[]}
46             else:
47                 result = {'code': '2003', 'message': '接口地址错误', 'data':
[]}
```

```
48         except Exception as error:#记录日志到 log.txt 文件
49             result={'code':'9999','message':'系统异常','data':[]}
50             logging.basicConfig(filename = config.src_path +
'/log/syserror.log',level =
logging.DEBUG,format='%(asctime)s %(filename)s[line:%(lineno)d] %(levelna
me)s %(message)s')
51             logger = logging.getLogger(__name__)
52             logger.exception(error)
53         return result
```

第 27~53 行代码定义了处理 POST 类型请求的方法。_ _http_post 是一个私有方法，只能在该类中调用，不能在类的外部使用。如需在类的内部调用，则用 self._ _http_post 调用。

- 第 37 行代码：调用类中的_ _new_param 方法来处理接口请求参数。在类中方法之间调用时，需加上 self。
- 第 38 行代码：使用 requests.post 方法处理 POST 类型数据。请求参数分别是接口地址、头文件信息、请求参数、verify 参数。其中，verify 参数是布尔类型，默认为 True，即启动 HTTPS 的 SSL 证书验证。此处不需要，所以将其设置为 False。
- 第 39 行代码：使用 response.status_code 参数判断该接口请求是否成功。此处的返回码就是正常的 HTTP 状态码。一般而言，200 代表响应正常，即服务器端对本次请求返回了数据包，此时并不知道返回的数据包是什么样的。
- 第 40、41 行代码：在 response.status_code==200 情况下，durtime 是接口发出请求到收到响应的时间差，可以作为 result 值的补充。其后，result 的数据是一个字典，而这个字典的 data 就是返回包数据，使用 response.text 来获取。
- 第 42~53 行代码：对于其他情况（包括异常情况）都返回一个完整结构的字典文件，调用者通过 result 中的 code 和 message 参数值来做进一步处理。

代码：request.py

```
54     # GET 请求，参数在接口地址后面
55     def _ _http_get(self,interface_url,headerdata,interface_param):
56         '''
57         :param interface_url: 接口地址
```

```
58         :param headerdata: 请求头文件
59         :param interface_param: 接口请求参数
60         :return:字典形式结果
61         '''
62         try:
63             if interface_url != '':
64                 temp_interface_param = self._ _new_param(interface_param)
65                 if interface_url.endswith('?'):
66                     requrl = interface_url+temp_interface_param
67                 else:
68                     requrl = interface_url +'?'+ temp_interface_param
69                 response = requests.get(url=requrl,
headers=headerdata,verify=False,timeout=10)
70                 #print response
71                 if response.status_code==200:
72                     durtime = (response.elapsed.microseconds) / 1000  # 发
起请求和响应到达的时间，单位 ms
73                     result={'code':'0000','message':'成功
','data':response.text}
74                 else:
75                      result={'code':'3004','message':'接口返回状态错误
','data':[]}
76             elif interface_url == '':
77                 result={'code':'3002','message':'接口地址参数为空
','data':[]}
78             else:
79                 result={'code':'3003','message':'接口地址错误','data':[]}
80         except Exception as error:#记录日志到 log.txt 文件
81             result={'code':'9999','message':'系统异常','data':[]}
82             logging.basicConfig(filename = config.src_path +
'/log/syserror.log',level =
logging.DEBUG,format='%(asctime)s %(filename)s[line:%(lineno)d] %(levelna
me)s %(message)s')
83             logger = logging.getLogger(_ _name_ _)
84             logger.exception(error)
85         return result
```

第 54~85 行代码定义了处理 GET 类型请求的方法。该方法也是一个私有方法，不能

在类的外部调用。

- 第 65~68 行代码：由于 GET 类型请求的数据一般跟在 URL 地址之后，所以需要在 URL 地址和请求参数之间做一次拼接，得到发送请求的 URL 地址。这里增加了一次判断，使用 endswith()来判断 URL 地址是不是以问号结尾。如果是以问号结尾，则直接拼接上参数；如不是，则在加上问号后拼接上参数。
- 其他行代码：其他数据的处理和 POST 方式没有区别，读者可以参考 POST 方式中的代码解释来理解。

代码：request.py

```
 86     # 统一处理 HTTP 请求
 87     def http_request(self,interface_url,headerdata,
interface_param ,request_type):
 88         '''
 89         :param interface_url: 接口地址
 90         :param headerdata: 请求头文件
 91         :param interface_param: 接口请求参数
 92         :param request_type:请求类型
 93         :return:字典形式结果
 94         '''
 95         try:
 96            if request_type=='get' or request_type=='GET':
 97               result=self.__http_get(interface_url,headerdata,
interface_param)
 98            elif request_type=='post' or request_type=='POST':
 99               result=self._ _http_post(interface_url,headerdata,
interface_param)
100             else:
101                result={'code':'1000','message':'请求类型错误
','data':request_type}
102         except Exception as error:#记录日志到 log.txt 文件
103            result={'code':'9999','message':'系统异常','data':[]}
104            logging.basicConfig(filename = config.src_path +
'/log/syserror.log',level =
logging.DEBUG,format='%(asctime)s %(filename)s[line:%(lineno)d] %(levelna
me)s %(message)s')
```

```
105             logger = logging.getLogger(_ _name_ _)
106             logger.exception(error)
107         return result
```

第 86~107 行代码定义对外的接口集方法。该方法可以被其他代码调用，其内部就是调用上面两个私有方法。

- 第 87 行代码：定义 http_request 方法的入参。除常规的 3 个参数（请求地址、头文件、请求参数）外，还多了一个 request_type，该参数的值主要用来区别调用哪个私有方法。
- 第 96~101 行代码：第 96、98 这两行代码是依据 request_type 的值是 POST 还是 GET 来调用对应的私有方法。第 97、99 行代码则是调用私有方法，用来传递参数。在同一个.py 文件中调用一个类的其他方法时，需要使用“self.方法名”。
- 其他行代码：请参照前面的代码解释来理解。

代码：request.py

```
108  if _ _name_ _ == "_ _main_ _":
109      test_interface=RequestInterface()#实例化 HTTP 请求类
110      test_db=opmysql.OperationDbInterface(host_db='192.168.0.
104',user_db='root',passwd_db='root',name_db='test_interface',port_db=330
6,link_type=0)#实例化 MySQL 处理类
111      sen_sql="select exe_mode,url_interface,header_interface,
params_interface from case_interface where name_interface='getIpInfo.php'
and  id=1"
112      params_interface=test_db.select_one(sen_sql)
113      if params_interface['code']=='0000':
114          url_interface=params_interface['data']['url_interface']
115          temp=params_interface['data']['header_interface']
116          headerdata=eval(params_interface['data']['header_interface'])
#将 unicode 转换成字典
117          param_interface=params_interface['data']['params_interface']
118          type_interface=params_interface['data']['exe_mode']
119          if url_interface!='' and headerdata!='' and param_interface!=''
and type_interface!='':
```

```
120            result=test_interface.http_request(interface_url=url
_interface,headerdata=headerdata,interface_param=param_interface,request_
type=type_interface)
121            if result['code']=='0000':
122                result_resp=result['data']
123                test_db.op_sql("UPDATE case_interface SET
result_interface='%s' WHERE id=1" %result_resp)#将结果更新到case_interface
表中
124                print("处理HTTP请求成功，返回数据是：%s" %result_resp)
125            else:
126                print("处理HTTP请求失败")
127        else:
128            print("测试用例数据中有空值")
129    else:
130        print("获取接口测试用例数据失败")
```

第 108~130 行代码很显然是调试的代码。在写完类、方法和函数后，都需要做一定量的单元测试，以保证其正确性。

- 第 109 行代码：实例化前面定义的 HTTP 请求类。在后面的使用中，以实例化的参数 test_interface 代表该类。
- 第 110 行代码：实例化数据库模块里的 OperationDbInterface 类，并初始化数据库连接的参数。在后面的使用中，以实例化的参数 test_interface 代表该类。
- 第 111、112 行代码：作为调试使用，从接口用例表中获取一条测试用例数据，sen_sql 参数就是待执行的查询语句。这部分需要一定的 SQL 操作能力，其后调用实例 test_db 下的 select_one 方法以获得单条数据。
- 第 120 行代码：在获取到单条用例中需要的参数后，调用实例 test_interface 发送 HTTP 请求，并获取结果。由于所有方法返回的数据都有相对固定的格式，所以在对返回数据进行处理时，都需要先判断 code 值，再依据 code 值去获取 data 值。

6.3　本章所涉及的 Python 语法

6.3.1　数据类型

Python 中最常用的数据类型有 3 种——元组、列表和字典。掌握这 3 种类型的数据后，便可以比较方便地操作数据了。

1．元组

（1）元组（tuple）是有序的列表，即元组的数据有不可见的下标。而这个下标，就是获取元组数据的一种方式。下标是从 0 开始，其表现形式如：(1,2,test)。

（2）元组的数据是相对固定的，不能增加、删除和修改，所以它的一个重要用途是，保存固定的、安全要求高的数据。一般而言，元组中的数据随着元组的建立而确定。

（3）元组是用小括号()括起来的，空的元组是()。1 个元素的元组比较特殊，定义为(1,)，加逗号是为了和数学公式中的计算小括号相区别。

（4）元组本身虽然不支持修改，但如果元组中的元素是可以修改的，比如，元组中的一个数据是列表，则可以修改这个列表的数据。

（5）元组的常用方法见表 6-3-1。

表 6-3-1　元组的常用方法

方　法	解　释
tuple[i]	获取元组的单个值，i 是下标
tuple[i,j]	获取元组的区间值
len(tuple)	计算元组数据的个数
tuple1+tuple2	合并元组
max(tuple)	返回元组中元素的最大值
min(tuple)	返回元组中元素的最小值
tuple(list)	将列表转换为元组
del tuple	删除元组

2．列表

列表 list 和元组一样，也是有序的集合，所以也有下标。当然，也是通过下标来获取指定的数据。下标是从 0 开始，其表现形式如：[1,2,test]。

列表数据是用中括号[]括起来的数据，可随时进行增加、删除、修改、查看等操作。

列表的常用方法见表 6-3-2。

表 6-3-2　列表的常用方法

方　法	解　释
list[i]	获取列表的单个值，i 是下标
list[i,j]	获取列表的区间值
len(list)	计算列表数据的个数
list.append(value)	在列表末尾添加元素
list.pop()	删除列表末尾元素，返回已删除的数据
list.pop(i)	删除列表指定下标元素，返回已删除的数据
list[i]=value	修改列表指定下标元素的值，即重新赋值
max(list)	返回列表中元素的最大值
min(list)	返回列表中元素的最小值
list(tuple)	将元组转换为列表
del list	删除列表

3．字典

字典 dict 是以键值对（key-value）形式存在的数据，是用大括号{}括起来的数据，具有极快的查找速度。其表现形式如：{'name':'Tom','score':99}。

字典数据是随时灵活可变的，支持增加、删除、修改、查找等操作。但字典的 key 必须是不可变的，即固定的数据。比如，可以使用字符串，但不能使用列表。

涉及字典操作时，其 key 必须是存在的。如果 key 不存在，则会抛出异常 keyError。

不允许同一个键出现两次。在创建字典时，如果同一个键被赋值两次，则前一个值会被覆盖。字典是无序的，所以在两次打印字典时，会发现字典中的数据顺序不一致。这是没关系的，因为字典并不需要下标。

字典的常用方法见表 6-3-3。

表 6-3-3　字典的常用方法

方　法	解　释
dict['key']	获取字典中指定 key 的值
dict['key']=value	更新字典中指定 key 的值（key 存在时）。在 key 不存在时是新增
del dict['key']	删除字典中指定 key 及其值
dict.clear()	删除字典内的所有元素
len(dict)	计算字典中元素的个数
str(dict)	将字典转换成字符串类型
dict.keys()	获取字典的所有 key，存储在列表中
dict.values()	获取字典的所有 value，存储在列表中
dict.items	获取字典的键值对，以元组的形式存储在列表中
Key in dict.keys()	判断字典中是否有指定的 key，返回布尔类型

6.3.2　方法与函数

1．方法、函数的区别

表 6-3-4 直观地显示了函数和方法的区别。

表 6-3-4　函数和方法的区别

说　明	函　数	方　法

定义	def function(param1,param2):
	return param1+param2
		class method(object):
		def function(self,param1,param2):
		return param1+param2

使用	result1=function(1,2)	test=method() result2=test.function(1,2)

下面从几点来阐述函数和方法之间的区别。

区别一：看上面有没有“人”。函数是“光杆司令”，上面没“人”。如果上面有“人”，则是方法。

区别二：上面没“人”的函数，可以直接使用。上面有“人”的方法，要先把类实

例化，然后在实例化名称下使用该方法。

区别三：参数、函数参数是“实打实”的，需要几个就是几个。方法的参数比较“虚”，总多要一个 self，但又不用。

但它们有一个共同的地方——参数的定义是相同的。这将在后面讲到。

2．方法、函数的传参

（1）默认参数。

所谓默认参数，就是在定义函数（方法）时预先设置一个参数的值。这个参数在调用函数（方法）时传不传值都无所谓，不传就用默认值，传了就用传入的值。这样的好处是，简化了一些不经常变动的参数及其值，简化了调用方传参的数量。表 6-3-5 是对参数的解释和举例。

表 6-3-5　参数的解释和举例

类　型	解　释	举　例
默认参数	函数（方法）的参数中有一个默认参数。如果不传入新的参数名称，则使用这个默认参数及其值。如果传入了新的值，则使用新传入的参数及其值	def function(param1,param2=2): 　　return param1+param2
		result1=function(1)
		result2=function(1,param2=2)
		result3=function(1,param2=3)

可变参数	函数的传参数量可以为若干个。传入多少个参数（包括 0 个），在函数的内部都是把这些参数组装成一个元组来处理	def function(*params):
		　　sum=0
		　　for n in params:
		　　　　sum=sum+n
		　　return sum

		result1=function()
		result2=function(1)
		result3=function(1,2,3)
		result4=function(*(1,2,3))

续表

类　型	解　释	举　例
关键字参数	函数的传参数量可以为若干个，不管传入多少个参数（包括 0 个），函数内部都是把这些参数组装成一个字典来处理	def function(**params): return params result1=function() result2=function(name='zhangsan',score=90) result3=function(**{'name':'zhangsan','socre':90})

设置默认参数时，有几点要注意：

- 默认参数在参数顺序中排在最后，必选参数排在前面。
- 当函数有多个参数时，把变化大的参数放在前面，变化小的参数放在后面。变化小的参数就可以作为默认参数。

默认函数的调用方法如下。

比如定义的函数是 function(param1,param2=2,param3=3)，则有两种调用方法：

- 按照顺序提供默认参数。调用上面函数的方法是 function(1,2)。
- 不按照顺序提供默认参数。但是这样需要把参数名写上。调用上面函数的方法是是 function(1,param3=4)。

（2）可变参数。

可变参数，即该参数是可变的。“可变”是指能接受任意多个参数。函数（方法）内部都会将这些参数组装成一个元组来处理，所以一般都使用可变参数的函数（方法），其内部是对该可变参数的组合进行循环操作。

如要将一个列表或元组中的数据作为参数传递给可变参数，难道需要按序取出数据吗？在传递该列表（元组）时，只需在前面加一个*号，则可变参数就知道如何处理了。

表 6-3-5 中有可变参数的解释和举例。请读者运行表 6-3-5 中的例子，检查和理解其返回结果。

（3）关键字参数。

可变参数允许传入 0 个或任意个参数。这些可变参数，在函数调用时会自动被组装为一个 tuple。关键字参数也允许传入 0 个或任意个含参数名的参数，但这些关键字参数在函数内部会自动被组装为一个 dict。这就是关键字参数和可变参数的区别。从参数名称上也能看出端倪，关键字参数肯定是对字典来说的。

同样，关键字参数也支持参数是字典，但是需要在字典前加**。

3．方法、函数的参数组合

在 Python 的传参中，有必选参数、默认参数、可变参数、关键字参数。当然，这 4 种参数也可以组合在一起使用。但这 4 类参数是有顺序要求的，即：必选参数→默认参数→可变参数→关键字参数。在这几种参数同时存在的情况下，建议按以下实例的顺序传参。

```
def function(a,b,c=0,*param1,**param2):
pass
```

6.3.3 切片

切片是为方便获取区域数据而准备的。虽然使用循环的方式也可以获取区域数据，但其代码量也会增加，而引入切片则可以很方便地切割到需要的区域数据。关于切片的代码操作详见表 6-3-6。

表 6-3-6 切片

解　　释	方　　法
就像是用刀切东西一样，两刀下去，取中间的一段（即片）L=[1,2,3,4,5]	1. 切片是相对列表（List）而言的，需要知道列表索引： （1）列表（有 *N* 个元素）的正向索引是 0~(*N*–1)； （2）L（起始索引:终止索引:变化量），其中变化量可不写，默认等于 1
	2. 截取第 1~3 位：L[0:2]
	3. 截取全部（复制列表）：L[:]
	4. 截取第 1~4 位，间隔两个数截取：L[0:3:2]
	5. 截取第 3 个之后所有：L[2:]
	6. 截取第 3 个之前所有：L[:2]

请读者运行上面的例子，检查和理解其返回结果。

6.3.4 日志模块 logging

本书中使用最直接的日志模块 logging 来记录日志（详见下面的日志记录代码）。但其不足之处是，在每个需要日志记录的地方都要添加多行代码，不方便统一修改。

先来分析下面这个日志记录代码。

```
except Exception as error:  # 记录日志到../log/syserror.log 文件
   logging.basicConfig(filename=config.base_dir+     '/log/syserror.log',
level=logging.DEBUG,
format='%(asctime)s %(filename)s[line:%(lineno)d] %(levelname)s %(message
)s')
   logger = logging.getLogger(_ _name_ _)
   logger.exception(error)
```

Logging 模块是 Python 自带的日志记录模块，不需要安装，只是在使用前需要导入该模块。

（1）logging.basicConfig 是日志输出的基本配置，包括日志输出的地址、等级和格式，所以本处的 basicConfig 中主要是 3 个字段值。

- Filename：日志的文件名称。该处使用“路径+文件名称”来唯一标识文件。其中，config.base_dir 是 config.py 模块中的一个变量值，该值获取的是 config 文件所在路径，再拼接上 log 文件下的 syserror.log 目录（该目录固定记录系统错误日志）。
- Level：日志的最低等级。比该等级高的都要记录，低的则不需要记录。等级排序：CRItical > Error > Warning > Info > Debug。该处设置 Level 为 debug，则所有的错误信息都会被记录下来。如将 Level 设为 Info，则会打印 CRItical、Error、Waring、Info 等级错误。
- Format：代表记录在日志文件中的具体信息。其在 logging 模块中有规范要求，使用者只要加入自己需要的参数和组合方式，即可得到想要的日志形式。

（2）logging.getLogger(_ _name_ _)返回的是 logger 对象，而_ _name_ _代表当前 py 文件名称。

（3）logger.exception(error)记录当前异常信息，传入的参数名称是 error。

6.4 补充知识点

6.4.1 Python 的循环机制

循环就是让计算机做重复且有效的事情。在达到一定条件时，结束循环或者提前退出循环。Python 中的循环有两种：一种是用得最多的 for…in 关键字，另一种是 while 关键字。

下面的 for…in 循环体，用通俗的语言解释就是：循环 param 这个参数，其值在(1,2,3)这个元组中。循环的意思是逐个取出，之后打印出来。所以，其结果很明显就是 1、2、3。

```
 # -*- coding: utf-8 -*-
#_ _author_ _ = '大婶 N72'
for param in (1,2,3):
    print(param)
```

循环的意思是“逐个取出”，如要满足循环条件，则 in 后面必须是一个迭代体。所以，可以使用以下这些迭代体来做 for 循环的条件，比如列表[1,2,3]、range(m)、"test"等。

这里提到了函数 range(m)，该函数是 Python 自带的函数，可以生成一个整数序列。其源码是 range(start=None, stop=None, step=None)，start 代表开始值，stop 代表结束值，step 代表步幅（1 代表“逐个”，2 代表“间隔 1 个”）。下面举例说明：

```
 # -*- coding: utf-8 -*-
#__author__ = '大婶 N72'
#1~7 逐个输出，不包含最后一个
print(range(1,7,1))
#1~7 间隔输出，不包含最后一个
print(range(1,7,2))
```

6.4.2 logging

1．Logging 下的常用函数

- Logger.setLevel()：设置日志级别。
- Logger.addHandler() 和 Logger.removeHandler()：添加和删除一个 Handler。
- Logger.addFilter()：添加一个 Filter，起过滤作用。
- Logging.Handler：Handler 基于日志级别对日志进行分发。

2．format 常用格式

- %(levelno)s：打印日志级别的数值。
- %(levelname)s：打印日志级别名称。
- %(pathname)s：打印当前执行程序的路径，相当于 sys.argv[0]。
- %(filename)s：打印当前执行程序名。
- %(funcName)s：打印日志的当前函数。
- %(lineno)d：打印日志的当前行号。
- %(asctime)s：打印日志的时间。
- %(thread)d：打印线程 ID。
- %(threadName)s：打印线程名称。
- %(process)d：打印进程 ID。
- %(message)s：打印日志信息。

7

用 Python 处理 HTTP 返回包

本章讲什么：

1. 用 Python 如何处理 HTTP 返回包；
2. 本章涉及的 Python 部分语法。

本章介绍用 Python 处理 HTTP 返回包。通过第 6 章的学习，我们已经知道如何发送 HTTP 请求，并得到返回包。但是，只得到返回包并不是最终目的，处理返回包数据并得到自己想要的结果才是终极目标，这也是一次接口测试完成的标志。所以说，本章算是对前面所有 HTTP 内容的总结。学完本章后，读者应该可以独立处理 HTTP 接口，也可以扩展和修改代码。

7.1　提前工作

在开始学习本章内容之前，需要确认以下工作是否已经完成：

（1）是否已经新建接口测试数据库、对应的表，以及测试的初始化数据。当然，复杂的测试数据可以在后面再添加，这里只需在表中有可用的测试数据。

（2）能否使用 Python 代码操作 MySQL 数据库中的数据，实施增加、删除、修改、查询操作。

（3）能否使用 Python 结合数据库发送 HTTP 请求，并获取返回包数据。

如果上面的操作已经没问题了，则可以开始本章的学习。

7.2　处理 HTTP 返回包实例

7.2.1　用 Python 处理 HTTP 返回包的基础

对 HTTP 返回包数据的处理，没有一个统一的标准，一般是测试人员按照实际的需要来制定测试标准。作者总结了该类接口的大部分测试场景，归纳出对于 HTTP 接口返回包数据应检查以下两点：

（1）指定的某一个关键参数的值是否正确。

（2）指定的某些参数名是否都存在。

那么如何理解以上所说的这两点呢？下面通过具体的实例来解释。

```
{
    "id" : 90,
    "name" : "Python",
    "url" : "http://www.v2ex.com/go/Python",
    "title" : "Python",
    "title_alternative" : "Python",
    "topics" : 8306,
    "stars" : 5398,
    "header" : "这里讨论各种 Python 编程话题，也包括 Django、Tornado 等框架的讨论。这里是一个能够帮助你解决实际问题的地方。",
    "footer" : null,
    "created" : 1278683336,
    "avatar_mini" : "//v2ex.assets.uxengine.net/navatar/8613/985e/90_mini.png?m=1513063513",
    "avatar_normal" : "//v2ex.assets.uxengine.net/navatar/8613/985e/90_normal.png?m=1513063513",
    "avatar_large" : "//v2ex.assets.uxengine.net/navatar/8613/985e/90_large.png?m=1513063513"
}
```


上面的 JSON 数据正是访问地址 https://www.v2ex.com/api/nodes/show.json?name=Python 时（见第 2 章）获取到的返回包数据，具体可参考图 2-1-2。这里带着问题对这个返回包数据做一次分析。

1. 作为调用者，应怎样判断这个返回包数据的正确性？

可以从以下四方面判断：

（1）在接口协议文档中，该接口返回的数据结构是否正确。

（2）在接口协议文档中，该接口的返回包数据中是否包含指定的参数。

（3）在接口协议文档中，该接口的关键参数值是否正确。

（4）接口返回包数据是否符合业务逻辑要求。

2. 如何将手工测试的检查点转换成自动化测试的检查点？

在进行转换之前，有一点必须注意：手工测试的检查点不可能全部转换成自动化测

试的检查点，即使能转换也需要修改检查点。因为自动化测试是由代码完成的，所以，需要站在代码的角度去思考：

（1）返回包数据结构的正确性。这实际上是检查返回包数据的整体结构是否和接口协议文档中所要求的一致。比如，上面代码返回的是一个纯 JSON 格式的文件，如果接口文档要求该 JSON 的下一级还有目录，则返回包中就少了数据。本书未涉及返回包数据结构正确性的检查。

（2）接口返回包数据主要是供调用者使用的，那么“调用者所关心的若干个关键参数是否都存在”也是检查点之一，因为有时前端页面并不一定会处理接口返回包中参数缺失的情况。

（3）检查参数值是否正确。在实际工作中，应该检查若干个参数值是否正确。本书以检查一个参数值的正确性来举例。

（4）本书没有涉及功能自动化测试部分的内容，主要由于功能测试的逻辑性比较强，而实现自动化测试则性价比偏低。

下面是一个更复杂且在实际中比较常见的接口返回包数据，读者可以先思考一下这样的返回包数据该如何测试。

```
{
    "message": "获取附近服务商成功",
    "nextPage": 1,
    "pageNo": 0,
    "merchantInfos": [
        {
            "phone": "15100000000",
            "star": 5,
            "totalQualityEvaluation": 0,
            "photoUrls": "",
            "latitude": 0
        },
        {
            "phone": "15200000000",
            "detail": null,
```

```
            "sex": null,
            "serviceFrequency": 0
        }
    ],
    "resultCode": "000",
    "totalPage": 66746
}
```

7.2.2 用 Python 处理 HTTP 返回包的流程

1．关键参数值校验

图 7-2-1 所示的是用 Python 代码校验关键参数值的流程图。

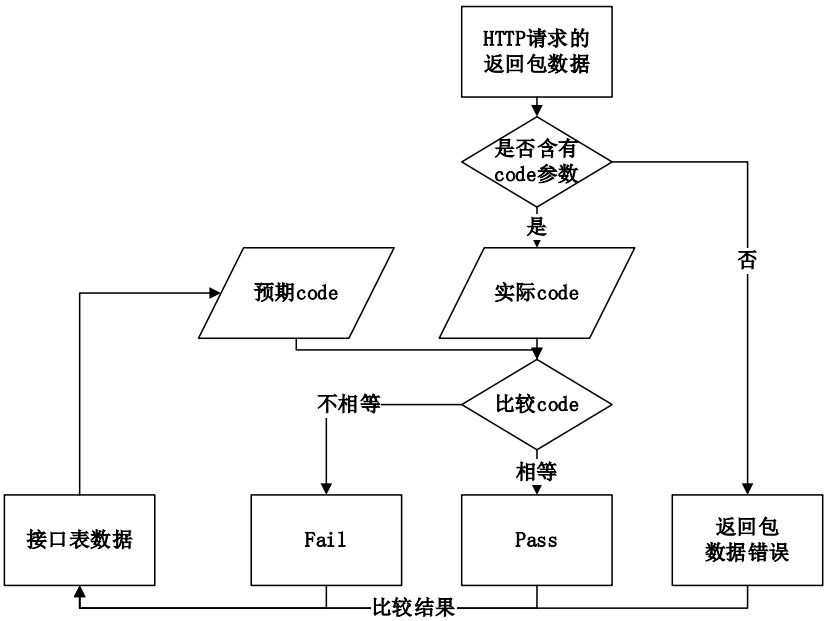


图 7-2-1　用 Python 代码校验关键参数值的流程图

对该流程图的解释如下：

（1）假设已经得到了一次 HTTP 请求的返回包数据，同时从数据库接口测试用例表（case_interface）中得到了该请求所要比较的关键参数的名称（code_to_compare 字段的值）。

（2）判断该返回包数据中是否有该关键参数。如不存在，则直接返回包错误；如存在，则获取该返回包中关键参数的值。

（3）从数据库接口测试用例表（case_interface）中得到了该请求所要比较的关键参数名称（code_to_compare 字段的值）的预期参数值（code_expect 字段的值）。

（4）将预期的关键参数的值与实际的关键参数的值做比较。如相等，则说明关键参数的值校验通过；如不相等，则说明关键参数的值校验失败。

（5）将比较的结果和 HTTP 返回包数据，重新写回接口测试用例表中的对应字段中，分别对应 result_code_compare 字段和 result_interface 字段。

2．参数完整性校验

图 7-2-2 所示的是用 Python 代码校验参数完整性的业务流程图。

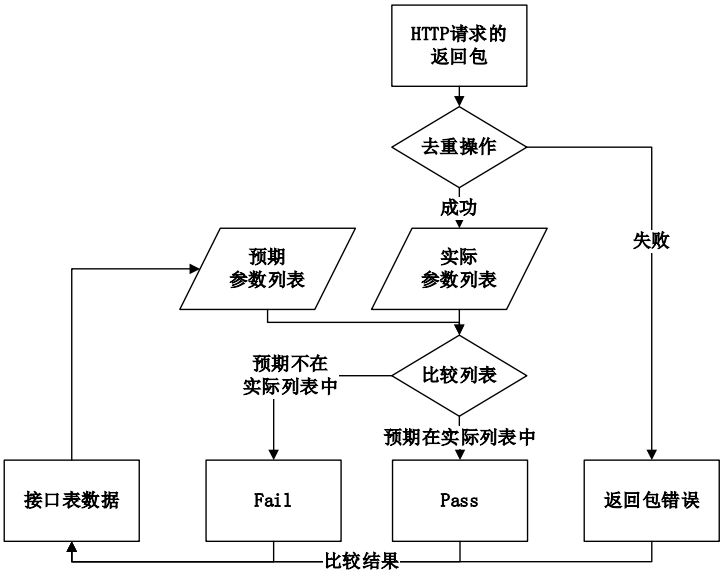


图 7-2-2　校验参数完整性的业务流程

对该流程图的解释如下：

（1）假设已经得到了一次 HTTP 返回包数据，现在要获取这个返回包中所有参数名的集合，自然要做“去重”处理。比如，在搜索接口返回包的列表数据中，很多字段都是相同的，需要提取“集合部分”组装成“列表”，这样就得到返回包的实际参数列表。

（2）如果“去重”操作失败，或者发生异常，则都作为返回包数据错误返回。

（3）从数据库接口测试用例表（case_interface）中，得到该请求所要比较参数的完整名称（params_to_compare 字段的值）的预期值。

（4）对预期的参数集合和 HTTP 返回包实际参数集合，采用“包含”关系来判断。如果预期的参数集合在实际参数集合中，则为成功，反之则为失败。

（5）将比较的结果和 HTTP 返回包实际参数集合，重新写回对应接口测试用例表中的字段（分别对应 result_params_compare 字段和 params_actual 字段）。

7.2.3 用 Python 处理 HTTP 返回包的代码

1．新建 compare.py 文件

在“common”目录下新建一个 compare.py 文件（当然，读者也可以按照自己的习惯来命名）。可以按照下面的步骤操作。

（1）右击 common 文件夹，在弹出的菜单中选择“New”→“Python file”命令，新建.py 文件，将“Name”设为“compare”，然后单击“OK”按钮，如图 7-2-3 所示。

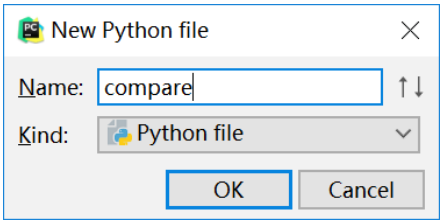


图 7-2-3 新建.py 文件

（2）新建.py 文件成功，目录结构如图 7-2-4 所示。

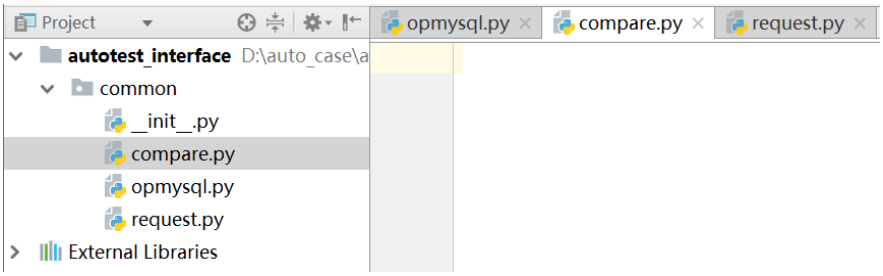


图 7-2-4　文件建成后的目录结构

2．在.py 文件中新建代码

代码：compare.py

```
1  # -*- coding:utf-8 -*-
2  #_ _author_ _ = '大婶 N72'
3  '''
4  定义数据比较方法
5  1.compare_param 是对外的参数比较类
6  2.compare_code 是关键参数值的比较方法，compare_params_complete 是参数完整性的
比较方法
7  3.get_compare_params 是获得返回包数据去重后集合的方法
8  4.recur_params 递归操作方法，辅助去重
9  '''
……
13  import json,os,logging
14  from common import opmysql
15  from public import config
16  operation_db=opmysql.OperationDbInterface()#实例化测试数据库操作类
17
18  class CompareParam(object):
19      #初始化数据
20      def _ _init_ _(self,params_interface):...
……
25      #定义关键参数值(code)比较
26      def compare_code(self,result_interface):...
……
62      #定义将接口返回数据中的参数名写入列表中
63      def get_compare_params(self,result_interface):...
……
```

```
 82      #参数完整性比较方法，比较传参值与_ _recur_params 方法返回的结果
 83      def compare_params_complete(self,result_interface):...
 ……
 113      #定义递归方法
 114      def _ _recur_params(self,result_interface):...
 ……
 136  #测试
 137  if _ _name_ _ == "_ _main__":
 138      sen_sql="select * from case_interface where name_interface=
'getIpInfo.php' and  id=1"
 139      params_interface=operation_db.select_one(sen_sql)
 140      result_interface=params_interface['data']['result_interface']
 141      test_compare_param=CompareParam(params_interface['data'])
 142      result_compare_code=test_compare_param.compare_code
(result_interface)#关键参数值比较
 143      print(result_compare_code)
 144      result_compare_params_complete=test_compare_param.
compare_params_complete(result_interface)#参数完整性比较
 145      print(result_compare_params_complete)
```

上面的代码是 request.py 文件的一部分，大致能看出这个文件的结构。具体每个方法是怎么实现的，会在下面逐步讲解。这里先把握整体的结构。

该文件封装的是一个 Python 处理 HTTP 返回包的类，类名是 CompareParam，类下面有 5 个方法。

- _ _init_ _：初始化数据。
- compare_code：关键参数的校验方法，能够被外部直接调用。
- compare_params_complete：参数完整性的校验方法。
- _ _recursive_params：递归方法，对返回包数据做去重处理，获得返回包数据的所有参数集合。
- get_compare_params：一个简单的获得返回包数据所有参数集合的方法，其作用与_ _recursive_params 相同。加入该方法是为了方便读者理解。在实际工作中都是直接使用_ _recursive_params 方法来处理返回包数据的。

以上都是按照需求来自定义的方法。如果读者有其他处理需求，可自行添加和修改。

最后在 if _ _name_ _ == "_ _main_ _"中写入的是实例化类和调用类中的方法。

下面是该.py 文件的全部代码。

代码：compare.py

```
1  # -*- coding:utf-8 -*-
2  #_ _author_ _ = '大婶 N72'
3  '''
4  定义数据比较方法
5  1.compare_param 是对外的参数比较类
6  2.compare_code 是关键参数值的比较方法，compare_params_complete 是参数完整性的
比较方法
7  3.get_compare_params 是获得返回包数据去重后集合的简单方法
8  4.recur_params 是递归操作方法，辅助去重使用
9  '''
10
11  import json,os,logging
12  from common import opmysql
13  from public import config
14  operation_db=opmysql.OperationDbInterface()#实例化测试数据库操作类
15
16  class CompareParam(object):
17      #初始化数据
18      def _ _init_ _(self,params_interface):
19          self.params_interface=params_interface #接口入参
20          self.id_case=params_interface['id']    #测试用 ID
21          self.result_list_response=[]           #定义用来存储参数集的空列表
22          self.params_to_compare=params_interface['params_to_compare']#
定义参数完整性的预期结果
```

下面是对上一段代码的解释。

- 第 14 行代码：实例化操作数据库的类，实例名为 operation_db。
- 第 16~22 行代码：定义参数比较的类，其名称为：CompareParam。在初始化方法中，接收的参数是接口参数，即数据库接口用例表（case_interface）中的一条条

测试数据；再对入参做以下处理，获取必要的参数，用于其他方法的调用。

代码：compare.py

```
24     def compare_code(self,result_interface):
25         '''
26         :param result_interface: HTTP 返回包数据
27         :return:返回码 code，返回信息 message，数据 data
28         '''
29         try:
30             if result_interface.startswith('{') and
isinstance(result_interface,str):
31                 temp_result_interface=json.loads(result_interface)#将字
符类型转换成字典类型
32                 temp_code_to_compare=self.params_interface
['code_to_compare']#获取待比较 code 的名称
33                 if temp_code_to_compare in temp_result_interface.keys():
34                     if
str(temp_result_interface[temp_code_to_compare])==str(self.params_interfa
ce['code_expect']):
35                         result={'code':'0000','message':'关键字参数值相同
','data':[]}
36                         operation_db.op_sql("UPDATE case_interface set
code_actual='%s',result_code_compare=%s where id=%s"
37                                             %(temp_result_interface[temp_cod
e_to_compare],1,self.id_case))
38                     elif str(temp_result_interface[temp_code_to_compare])!
=str(self.params_interface['code_expect']):
39                         result={'code':'1003','message':'关键字参数值不相同
','data':[]}
40                         operation_db.op_sql("UPDATE case_interface set
code_actual='%s',result_code_compare=%s where id=%s"
41                                             %(temp_result_interface[temp_cod
e_to_compare],0,self.id_case))
42                     else:
43                         result={'code':'1002','message':'关键字参数值比较出
错','data':[]}
44                         operation_db.op_sql("UPDATE case_interface set
code_actual='%s',result_code_compare=%s where id=%s"
```

```
 45                                          %(temp_result_interface[temp_cod
e_to_compare],3,self.id_case))
 46               else:
 47                   result={'code':'1001','message':'返回包数据无关键字参数
','data':[]}
 48                   operation_db.op_sql("UPDATE case_interface set
result_code_compare=%s where id=%s" %(2,self.id_case))
 49           else:
 50               result={'code':'1000','message':'返回包格式不合法
','data':[]}
 51               operation_db.op_sql("UPDATE case_interface set
result_code_compare=%s where id=%s" %(4,self.id_case))
 52        except Exception as error:#记录日志到 log.txt 文件
 53           result={'code':'9999','message':'关键字参数值比较异常
','data':[]}
 54           operation_db.op_sql("UPDATE case_interface set
result_code_compare=%s where id=%s" %(9,self.id_case))
 55           logging.basicConfig(filename = config.src_path +
'/log/syserror.log',level =
logging.DEBUG,format='%(asctime)s %(filename)s[line:%(lineno)d] %(levelna
me)s %(message)s')
 56           logger = logging.getLogger(__name__)
 57           logger.exception(error)
 58        finally:
 59           return result
```

第24~59行代码定义了关键参数值比较的方法。该方法接收的参数是result_interface，即返回包数据。

- 第30行代码：使用字符串的startswith方法来判断字符串是否以“{”开头且是字符串，返回包数据（JSON格式）是否能被转换成Python中的字典。如果不能，则返回一个封装好的 result 字典，同时更新数据库接口用例表。如将该条数据的result_code_compare字段的值设为4（4是数据库表定义时的值，用于标识该字段），则这个地方默认返回包数据的格式是JSON。当然，如果接口返回的是其他格式，比如XML格式，则此处需要稍做修改。

- 第 31、32 行代码：分别获取字典形式的返回包数据，并将其命名为 temp_result_interface。待比较的关键参数的名称是从数据库接口表中获取 code_to_compare 字段的值，并将其命名为 temp_code_to_compare。
- 第 33 行代码：判断关键参数是否在返回包的一级键中。如果在，则继续处理；否则直接执行第 47 行代码（返回 JSON 形式的 result，第 48 行代码更新数据库记录）。
- 第 34~37 行代码：获取 temp_result_interface 字典中 temp_code_to_compare 键的值，并与接口参数中的 code_expect 字段的值进行比较。如果两者相等，则返回 JSON 格式的 result，并更新数据库记录；否则按照第 8~45 行代码的两种情况进行处理。这里需要注意：在更新数据库记录时，result_code_compare 字段的值是与比较结果相关联的。

代码：compare.py

```
60      #定义将接口返回数据中的参数名写入列表的方法
61      def get_compare_params(self,result_interface):
62          '''
63          :param result_interface: HTTP 返回包数据
64          :return:返回码 code，返回信息 message，数据 data
65          '''
66          try:
67              if result_interface.startswith('{') and
isinstance(result_interface,str):
68                  temp_result_interface=json.loads(result_interface)
69                  self.result_list_response=temp_result_interface.keys()
70                  result={'code':'0000','message':'成功
','data':self.result_list_response}
71              else:
72                  result={'code':'1000','message':'返回包格式不合法
','data':[]}
73          except Exception as error:#记录日志到 log.txt 文件
74              result={'code':'9999','message':'处理数据异常','data':[]}
75              logging.basicConfig(filename = config.src_path +
'/log/syserror.log',level =
logging.DEBUG,format='%(asctime)s %(filename)s[line:%(lineno)d] %(levelna
```

```
me)s %(message)s')
 76            logger = logging.getLogger(__name__)
 77            logger.exception(error)
 78        finally:
 79            return result
```

下面是对上一段代码的解释。

第 60~79 行代码定义一个获取返回包数据中所有参数的方法，该方法只适用于只有一层结构的 JSON 格式数据。定义此方法主要是为了便于读者更好地理解如何获取返回包的参数，并将其存储在指定列表中，并不是实际使用的方法。

第 69 行代码：通过字典的 keys()方法获取该字典的一级键名称。该方法返回的恰好是一个列表格式键的集合，所以，通过该方法能成功地获得一级键名集合。

代码：compare.py

```
111     #定义递归方法
112     def __recur_params(self,result_interface):
113         #定义递归操作，将接口返回数据中的参数名写入列表中（去重)
114         try:
115             if result_interface.startswith('{') and
isinstance(result_interface,str):  # 入参是字符串类型，且能被转换成字典
116                 temp_result_interface=json.loads(result_interface)
117                 self.__recur_params(temp_result_interface)
118             elif isinstance(result_interface, dict):  # 入参是字典
119                 for param, value in result_interface.items():
120                     self.result_list_response.append(param)
121                     if isinstance(value,list):
122                         for param in value:
123                             self.__recur_params(param)
124                     elif isinstance(value,dict):
125                         self.__recur_params(value)
126                     else:
127                         continue
128             else:
129                 pass
130         except Exception as error:#记录日志到 log.txt 文件
131             logging.basicConfig(filename = config.src_path +
```

```
'/log/syserror.log',level =
logging.DEBUG,format='%(asctime)s %(filename)s[line:%(lineno)d] %(levelna
me)s %(message)s')
 132            logger = logging.getLogger(__name__)
 133            logger.exception(error)
 134            return {'code': '9999', 'message': '处理数据异常', 'data': []}
 135        return {'code':'0000','message':'成功
','data':list(set(self.result_list_response))}
```

上一段代码定义了获取复杂返回包参数集合的“递归”操作。所谓“递归”操作，就是在函数（方法）内部再调用该函数（方法）自己。请注意 7.2.1 小节结尾部分提到了一个复杂的返回包，该返回包内部是列表与字典的嵌套。对于此类返回包数据，如果用代码层层剥离显然是不现实的，但此类数据的类型无外乎 3 种：字典、列表和值。所以，只要定义了这几种参数的处理方法，则无论返回包数据多么复杂都可以用很少的代码去解决。

- 第 115~117 行代码先判断字符串是否能被转换成字典。如果能，则直接调用本方法进行递归，其中的参数就是转换成字典后的数据。
- 第 118~129 行代码：如果参数是字典，则按照字典的键和值分别进行处理。即循环获取字典的键和值，键名直接被存储在最终的列表中，而值还需要再做字典、列表和值的三层判断，并做对应的递归。
- 第 135 行代码：最终输出结果是字典形式的数据。其中，data 键的值是被做了一次转换处理，即用 set 方法对 self.result_list_response“去重”后再用 list 方法将其转换成列表。

代码：compare.py

```
 80     #定义参数完整性的比较方法，将传参值与_ _recur_params 方法返回结果进行比较
 81     def compare_params_complete(self,result_interface):
 82         '''
 83         :param result_interface:HTTP 返回包
 84         :return:返回码 code，返回信息 message，数据 data
 85         '''
 86         try:
 87             temp_compare_params=self.__recur_params(result_interface)#
```

```
获取返回包的参数集
 88            if temp_compare_params['code']=='0000':
 89                temp_result_list_response=temp_compare_params['data']#获
取接口返回参数去重列表
 90                if self.params_to_compare.startswith('[') and
isinstance(self.params_to_compare,str):#判断测试用例表中预期结果集是否为列表
 91                    list_params_to_compare=eval(self.params_to_compare)#
将数据库表中的 unicode 编码数据转换成原列表
 92                    if
set(list_params_to_compare).issubset(set(temp_result_list_response)):#判断
集合的包含关系
 93                        result={'code':'0000','message':'参数完整性比较一致
','data':[]}
 94                        operation_db.op_sql('UPDATE case_interface set
params_actual="%s",result_params_compare=%s where
id="%s"'%(temp_result_list_response,1,self.id_case))
 95                    else:
 96                        result={'code':'3001','message':'实际结果中元素不都
在预期结果中','data':[]}
 97                        operation_db.op_sql('UPDATE case_interface set
params_actual="%s",result_params_compare=%s where
id="%s"'%(temp_result_list_response,0,self.id_case))
 98                else:
 99                    result={'code':'4001','message':'用例中待比较参数集错误
','data':self.params_to_compare}
 100            else:
 101                result={'code':'2001','message':'调用_ _recur_params 方法
返回错误','data':[]}
 102                operation_db.op_sql('UPDATE case_interface set
result_params_compare=%s where and id="%s"'%(2,self.id_case))
 103        except Exception as error:#记录日志到 log.txt 文件
 104            result={'code':'9999','message':'参数完整性比较异常
','data':[]}
 105            operation_db.op_sql('UPDATE case_interface set
result_params_compare=%s where id="%s"' %(9,self.id_case))
 106            logging.basicConfig(filename = config.src_path +
'/log/syserror.log',level =
logging.DEBUG,format='%(asctime)s %(filename)s[line:%(lineno)d] %(levelna
me)s %(message)s')
```

```
107             logger = logging.getLogger(_ _name_ _)
108             logger.exception(error)
109         finally:
110             return result
```

第 80~110 行代码定义了获取参数完整性校验的方法，主要是借助前面的递归方法来获取返回包参数集合和预期参数集合，并做比较。

- 第 87 行代码：通过调用递归方法获取返回包中的数据集合。
- 第 90 行代码：判断接口用例表中待比较的参数集合。参数集合只能是列表形式。
- 第 91 行代码：用 eval()方法将字符串类型的列表转换成原列表，方便后面的比较计算。
- 第 92~97 行代码：使用集合的“包含”关系来比较两个列表数据的包含关系。所以，先使用 set()方法将对应的列表转换成集合，再使用集合的 issubset()方法来计算包含关系。该条语句的意思是，如果 list_params_to_compare 是 temp_result_list_response 的子集则结果为真。
- 第 93、94 行代码：在第 92 行代码的包含关系成立时，一方面返回字典形式的 result，另一方面更新数据库接口用例表（重点是更新 result_params_compare 字段的值）。

代码：compare.py

```
136  #测试
137  if _ _name_ _ == "_ _main_ _":
138      sen_sql="select * from case_interface where
name_interface='getIpInfo.php' and  id=1"
139      params_interface=operation_db.select_one(sen_sql)
140      result_interface=params_interface['data']['result_interface']
141      test_compare_param=CompareParam(params_interface['data'])
142      result_compare_code=test_compare_param.compare_code
(result_interface)#关键参数值比较
143      print(result_compare_code)
144      result_compare_params_complete=test_compare_param.
compare_params_complete(result_interface)#参数完整性比较
145      print(result_compare_params_complete)
```

下面是对上一段代码的解释。

- 第 138~140 行代码：从 case_interface 表中获取测试结果数据。
- 第 142~145 行代码：分别调用不同的方法测试结果的正确性。

7.3　本章所涉及的 Python 语法

7.3.1　json 方法

Python 的 json 模块中提供了一种很简单的方式来编码和解码 JSON 数据，其中两个主要的函数是 json.dumps()和 json.loads()。

- json.dumps()：将 Python 中纯粹的字典转换成 JSON 编码的字符串。
- json.loads()：与 dumps 方法相反，将 JSON 编码的字符串转换成 Python 中纯粹的字典。

JSON 格式文件、Python 中的字典、JSON 编码的字符串又是什么样子的呢？下面是三种数据类的举例：

- JSON 格式文件：{"phone":"18199990000","type":1}。
- Python 中的字典：{'phone':'18199990000','type':1}。
- JSON 编码的字符串：'{"phone":"18199990000","type":1}'。

下面的两段代码分别演示了 JSON 编码的字符串和字典数据之间的转换，请读者运行查看。

```
# -*- coding:utf-8 -*-
#将字典数据转换成 JSON 编码的字符串
import json
data = {'phone':'18199990000','type':1}
json_str = json.dumps(data)
print json_str
print type(json_str)
```

```
# -*- coding:utf-8 -*-
#将 JSON 编码的字符串转换成字典数据
import json
data = '{"phone":"18199990000","type":1}'
json_dict = json.loads(data)
print json_dict
print type(json_dict)
```

7.3.2 字典的两个方法

1．判断键是否存在于字典中

第 5 章介绍了如何判断键是否存在于字典中，当时用的是 has_key 方法，其实还有另一个方法：in/not in。前一种方法在 Python 3 中已经不再使用了，所以推荐大家使用 in/not in 方法，它更直观。举例如下：

```
# -*- coding:utf-8 -*-
#将字典数据转换成 JSON 编码的字符串
data = {'phone':'18199990000','type':1}
print(data.has_key('phone'))                    #Python 2 的判断方法
print('phone' in data)                          #Python 3 的判断方法
print('mobilephone' not in data)                #Python 3 的判断方法
```

通过上面例子的实测，读者能够理解这两种方法的不同之处：

- dict.has_key(key)用于判断字典中是否有 key。
- key in/not in dict 用于判断 key 是否在字典中。

2．字典键值迭代器

字典是由键值对组成的，那么如何获取所有的键值对数据呢？Python 3 提供.items()方法，该方法返回的是每一个键值对的元组。举例如下：

```
# -*- coding:utf-8 -*-
#将字典数据转换成 JSON 编码的字符串
data = {'phone':'18199990000','type':1}
print(data.items())                 #返回可遍历的(键,值)元组列表
for tuple_data in data.items():
```

```
    print(tuple_data)
for key,value in data.items():    #获取 key 和 value 元组的值
    print (key,value)
```

7.3.3　eval()与 instance()方法

eval()方法是将字符串当成有效的 Python 表达式来求值，并返回计算结果。本小节是将字符串类型的文件转换为本来面目（即实现 list、tuple、dict 和 string 之间的转换）。比如前面讲的 json.loads 方法，可以使用 eval()方法来替代它。举例如下：

```
# -*- coding:utf-8 -*-
#将 JSON 编码的字符串转换成纯粹的字典数据
import json
data = '{"phone":"18199990000","type":1}'
json_dict = eval(data)
print json_dict
print type(json_dict)
```

请读者通过运行以上代码来比较二者转换数据的差别。

isinstance()方法用于判断参数的类型是否在给定的类型之中，其支持多个类型，返回结果是布尔类型——True/False。举例如下：

```
# -*- coding:utf-8 -*-
#isinstance(参数,类型(支持多个))
import json
data = {'phone':'18199990000','type':1}
print isinstance(data,(dict,list))
print isinstance(data,list)
```

7.3.4　set()与 issubset()方法

Python 有三种常用数据类型——元组、列表和字典。下面要介绍的集合（set）是和列表（list）有相似功能的数据类型。本书代码使用集合数据类型，主要是利用它的唯一性（集合中的数据不重复）和包含关系（集合之间包含）。

1. 唯一性

用 set()方法将列表转换成集合，能将其中重复的数据去除，实现唯一性。例如下面的代码：

```
# -*- coding:utf-8 -*-
#用 set()方法去重
test_list=[1,2,3,2,4]
test_set=set(test_list)
print (test_set)
print list(test_set)
```

2. 包含关系

利用 issubset()方法来判断两个集合之间是否有包含关系。比如 s.issubset(t)，是判断 s 中的每一个元素是否都在 t 中，如果是，则返回 True，否则返回 False。例如下面的代码：

```
# -*- coding:utf-8 -*-
#用 issubset()判断包含关系
print set([1,2,3,4]).issubset(set([1,2]))
print set([3,4]).issubset(set([1,2,3,4]))
```

7.4 补充知识点

7.4.1 Python 的垃圾回收机制

Python 的 GC 模块主要通过“引用计数”（reference counting）来跟踪和回收垃圾。在引用计数的基础上，可以通过“标记-清除”（mark-sweep）来解决容器对象可能产生的循环引用问题，还可以通过“分代回收”（generation collection）来实现“以空间换取时间”，进一步提高垃圾回收的效率。

1．引用计数

在 Python 中，大多数对象的生命周期都是通过对象的引用计数来管理的。从广义上来讲，引用计数也是一种垃圾回收机制，而且是一种最直观、最简单的垃圾回收机制，

其原理如下：

（1）当一个对象的引用被创建或者复制时，对象的引用计数加 1。

（2）当一个对象的引用被销毁时，对象的引用计数减 1。

（3）如果对象的引用计数减少为 0，则意味着对象没被任何人使用了，可以将其所占用的内存地址释放。

必须在每次分配和释放内存时加入管理引用计数的动作。与其他主流的垃圾回收机制相比，引用计数有一个最大的优点——实时性：任何内存地址，一旦没有指向它的引用，就会立即被回收。而其他的垃圾回收机制，必须在某种特殊条件下（比如内存分配失败）才能进行无效内存的回收。

2．标记-清除

“标记-清除”机制是为了解决循环引用的问题，可以包含其他对象引用的容器对象（比如：list、set、dict、class、instance 都可能产生循环引用）。

我们必须要承认一个事实：如果两个对象的引用计数都为 1，而它们之间仅仅存在循环引用，则这两个对象都需要被回收。因为，虽然它们的引用计数表现为非 0，但实际上有效的引用计数为 0。如果先将循环引用去掉，那么这两个对象的有效计数就“现身”了。假设有两个对象 A 和 B，从 A 出发，因为它有一个对 B 的引用，则将 B 的引用计数减 1；然后顺着引用到达 B，因为 B 有一个对 A 的引用，同样将 A 的引用减 1。这样就可以去除循环引用对象之间的“环”。

“标记-清除”机制采用了更好的做法：并不改动真实的引用计数，而是将集合中对象的引用计数复制一份副本，改动该对象引用的副本。对于副本所做任何的改动，都不会影响对象生命周期的维护。

3．分代回收

“标记-清除”机制带来的额外操作与系统中内存块的数量相关：需要回收的内存块越多，则垃圾检测需要的额外操作就越多，而垃圾回收需要的额外操作就越少；反之，

需要回收的内存块越少，则垃圾检测需要的额外操作就越多，而垃圾回收需要的额外操作就越少。

为了提高垃圾回收的效率，可采用“以空间换时间”策略。该策略的原理是：将系统中的所有内存块根据其存活时间划分为不同的集合，每一个集合是一“代”。垃圾回收的频率随着“代”的存活时间的增大而减小。即活得越长的对象，就越不可能是垃圾，就应该减少对它的垃圾回收频率。

如何衡量这个存活时间呢？通常可以这样来衡量：如果一个对象经过的垃圾回收次数越多，则可以得出该对象存活的时间越长。

7.4.2 字符串的 startswith()和 endswith()方法

1．startswith() 方法

startswith()方法用于检查字符串是否以指定子字符串开头。如果是，则返回 True，否则返回 False。该方法有 beg 和 end 两个参数，用于在指定范围内进行检查，见下面的代码：

```
str.startswith(str, beg=0,end=len(string));
```

- str：要检测的字符串。
- beg：可选参数，用于设置字符串检测的起始位置。
- end：可选参数，用于设置字符串检测的结束位置。

2．endswith() 方法

endswith() 方法用于判断字符串是否以指定后缀结尾。如果是，则返回 True，否则返回 False。见下面的代码：

```
str.endswith(suffix[, start[, end]])
```

- suffix：可以是一个字符串或一个元素。
- start：字符串的开始位置。

- end：字符串的结束位置。

备注：start 从 0 开始。end 是结束，但不包括最后一个字符串。

请读者练习下面的代码，以加深对 startswith()和 endswith()方法的理解。

```
# -*- coding:utf-8 -*-
#__author__ = '大婶 N72'
str = "this is string example....wow!!!"
print (str.startswith( 'this' ))
print (str.startswith( 'string', 8 ))
print (str.startswith( 'i', 2, 3 ))
print (str.endswith( '!!!' ))
print (str.endswith( 'h', 0, 1 ))
```

8

用 Python 导出测试数据

本章讲什么：

1. 用 Python 将数据库表中的数据导出到 Excel 中；
2. 本章所涉及的 Python 部分语法；
3. 整体业务流程图分析。

我们在给领导汇报测试结果时是不可能用代码进行演示的，所以需要将结果数据导出并将进行图表化。本书只涉及将测试结果数据单纯地导出到 Excel 中，不涉及生成图标这部分内容。

8.1 提前工作

本章与前面几章没有特别大的关联，只要能导出指定的接口测试用例表中的数据即可。即使没有测试结果，读者也可以自己完善接口测试用例表，并使用本章的代码来导出数据。当然，希望读者能够按照本章的内容完成真正的接口请求和数据包处理，并在此基础上将数据导出到 Excel 中。

1．准备初始化数据

在 config_total 表中新增导出数据设置：

```
 INSERT INTO 'config_total' ('key_config', 'value_config', 'description',
'status',    'create_time',    'update_time')    VALUES    ('name_export',
"['getIpInfo.php','BaikeLemmaCardApi']", '导出接口数据配置',
 '1',NOW(),NOW());
```

该段 SQL 语句的解释是：key_config 值被固定为 name_export，代表该类数据是用来导出的；value_config 值是以列表形式存放待导出接口的名称，其每个值对应的是 case_interface 表中的接口名称。

2．在 config.py 文件中新增初始化数据

具体语法如下：

```
 field_excel=['编号','接口名称','用例级别','请求类型','接口地址','接口头文件','接
口请求参数','接口返回包','待比较参数','实际参数值','预期参数值','参数值比较结果','待
比较参数集合','实际参数集合','参数完整性结果','用例状态','创建时间','更新时间']#导出
的 excel 表格标题
```

3．新建 report 文件夹

在工程目录下新建一个 report 文件夹，用于存储测试报告的数据。

4．新建一个空的 Excel 文件

在 report 文件夹下新建一个空的 Excel 文件，并命名为 report_module.xls。注意，本代码所使用的模块暂且只支持 Excel 2007 及之前的版本。如果读者想操作.xlsx 文件，则可以使用 openpyxl 模块，它用于给报告文件提供可复制的模板，其内部结构如图 8-1-1 所示。

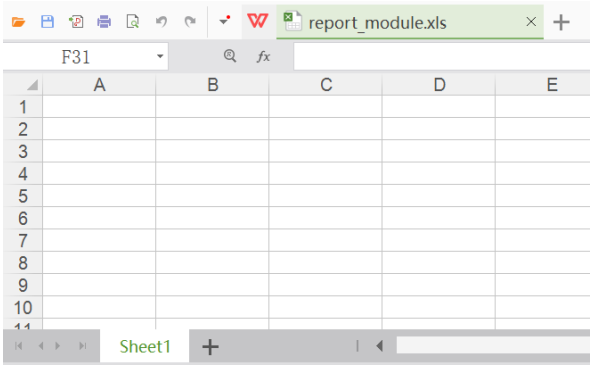


图 8-1-1　空的测试报告模板

8.2　用 Python 导出测试数据

8.2.1　导出测试数据的基础知识

导出测试报告的思路如下：

（1）从配置文件中获取要导出的接口数据。

（2）从接口测试用例表中获取待导出接口的有效测试数据。

（3）复制一份模板报告文件，将其重新命名作为本次导出的报告。

（4）在新的 Excel 文件中新建一张工作表，以各个接口的名称命名。

（5）将获得的相关数据写入报告文件的对应工作表中并保存。

8.2.2　导出测试数据实例

1．新建 analyse.py 文件

在 common 目录下新建一个以.py 为后缀的文件，具体步骤如下所示。

（1）在 common 文件夹上单击鼠标右键，在弹出的快捷菜单中选择“New”→“Python file”命令，新建.py 文件，将“Name”设为“analyse”，单击“OK”按钮，如图 8-2-1 所示。

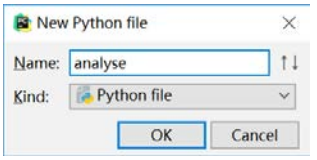


图 8-2-1　新建 analyse.py 文件

（2）新建.py 文件成功。其目录结构如图 8-2-2 所示。

图 8-2-2　创成后的目录结构

2．在.py 文件中创建代码

代码：analyse.py

```
# -*- coding:utf-8 -*-
#__author__ = '大婶 N72'
'''
定义导出数据库表数据到 Excel 的类。如果要使用 Excel 2007 版本，则使用 openpyxl 模块
1.用 init 方法初始化获取配置文件数据
2.export2excel 为主方法
'''
from xlrd import open_workbook
from xlutils.copy import copy
```

```
10  import os,logging,datetime
11  from public import config
12  from common import opmysql
13  operation_db=opmysql.OperationDbInterface(link_type=1)#实例化自动化测试
数据库操作类
14  class AnalyseData(object):
15      '''
16      定义对接口测试数据进行分析的类，包含的方法有：
17      1.导出测试数据到 Excel 中
18      '''
19      def __init__(self):...
……
21      #定义导出数据方法
22      def export2excel(self,names_export):...
……
61  if _ _name_ _ == "_ _main_ _":
62      names_export=operation_db.select_one("select value_config from config_
total where status=1 and key_config='name_export'")#获取导出的接口数据元组
63      if names_export['code']=='0000':            #判断查询结果
64          temp_export=eval(names_export['data'][0])#获取查询数据，并将其转换成字典
65          test_analyse_data=AnalyseData()
66          result_export=test_analyse_data.export2excel(temp_export)#导出结果
67          print (result_export)
68      else:
69          print('获取导出接口集失败')
```

上面是 analyse.py 文件的一部分，大致能看出这段代码的结构。具体每个方法怎么实现，会在下面部分逐步地讲解。这里先把握整体的结构。

这段代码封装的是导出数据到 Excel 中的类，类名是 AnalyseData。

在最后的 if _ _name_ _ == "_ _main_ _"中写入的是实例化类和调用类中的方法。

下面是该 py 文件的全部代码。

代码：analyse.py

```
1  # -*- coding:utf-8 -*-
2  #_ _author_ _ = '大婶 N72'
3  '''
4  定义导出数据库表中的数据到 Excel 中的类。如要使用 Excel2007 版本，则使用 openpyxl
```

```
模块
 5  1.用 init 方法初始化获取配置文件数据
 6  2.export2excel 为主方法
 7  '''
 8  from xlrd import open_workbook
 9  from xlutils.copy import copy
 10  import os,logging,datetime
 11  from public import config
 12  from common import opmysql
 13  operation_db=opmysql.OperationDbInterface(link_type=1)#实例化自动化测试
数据库操作类
 14  class AnalyseData(object):
 15      '''
 16      定义对接口测试数据进行分析的类，包含的方法有：
 17      1.导出测试数据到 Excel 中
 18      '''
 19      def __init__(self):
 20          self.field= config.field_excel#初始化配置文件
```

下面是对上一段代码的解释。

- 第 8 行代码：从 xlrd 模块中导入 open_workbook 方法，主要用于打开指定的 Excel 文件。
- 第 9 行代码：从 xlutils.copy 模块中导入 copy 方法，用于复制指定目录下的 Excel 文件。
- 第 13 行代码：实例化数据库操作的类 OperationDbInterface。注意传参中的 link_type=1，按照设计，查询的结果应以元组形式存在，以便于下面的数据展示。
- 第 19、20 行代码：初始化数据，获取 config.py 文件中的配置项值 field_excel。

代码：analyse.py

```
 21      #定义导出指定数据到 Excel 中的类
 22      def export2excel(self,names_export):
 23          '''
 24          :param names_export: 待导出的接口名称，列表形式
 25          :return:
 26          '''
 27          counts_export=len(names_export)     #导出总数
 28          fail_export=[]                      #导出失败接口名列表
```

```
 29        try:
 30            src = open_workbook(config.src_path + '/report/report_module.
xls',formatting_info=True)
 31            destination = copy(src)
 32            dt=datetime.datetime.now().strftime("%Y%m%d%H%M%S")#当前时间戳
 33            filepath=config.src_path+'/report/'+str(dt)+'.xls'
 34            destination.save(filepath)#保存模板表格到新的目录下
 35            for name_interface in names_export:
 36                cases_interface=operation_db.select_all("select * from
case_interface where case_status=1 and name_interface='%s'"
 37                                                        %(name_interface))#获取
指定接口的测试用例数据
 38                if len(cases_interface['data'])!=0 and cases_interface
['code']=='0000':
 39                  src = open_workbook(filepath,formatting_info=True)
 40                  destination = copy(src)
 41                    sheet = destination.add_sheet(name_interface,cell_
overwrite_ok=True)
 42                  for col in range(0,len(self.field)):
 43                      sheet.write(0,col,self.field[col])# 获取并写数据段
信息到 Sheet 中
 44                  for row in range(1,len(cases_interface['data'])+1):
 45                     for col in range(0,len(self.field)):
 46
sheet.write(row,col,'%s' %cases_interface['data'][row-1][col])#写数据到对应
Excel 表中
 47                  destination.save(filepath)
 48              elif len(cases_interface['data'])==0 and cases_interface
['code']=='0000':
 49                  fail_export.append(name_interface)
 50              else:
 51                  fail_export.append(name_interface)
 52            result={'code':'0000','message':'导出总数：%s，失败数：%s'
 53                                            %(counts_export,len(fail_export)
),'data':fail_export}
 54        except Exception as error:#记录日志到 log.txt 文件
 55            result={'code':'9999','message':'导出过程异常|导出总数：%s，失败
数：%s'
 56                                            %(counts_export,len(fail_export)
),'data':fail_export}
```

```
 57                    logging.basicConfig(filename = config.src_path +
'/log/syserror.log',level                                              =
logging.DEBUG,format='%(asctime)s %(filename)s[line:%(lineno)d] %(levelna
me)s %(message)s')
 58            logger = logging.getLogger(__name__)
 59            logger.exception(error)
 60        return result
```

下面是对上一段代码的解释。

- 第 22 行代码：定义导出数据到 Excel 中的方法 export2excel，其接收的参数是待导出的接口名称列表。如要导出 3 个接口，则需要在数据库配置表中添加导出配置数据，即 8.2 节中提到的 config_total 表中 key_config 值为 name_export 的数据。
- 第 27、28 行代码：在该方法下先定义了两个参数，一个是需要导出的接口总数（计算出 names_export 列表的长度即可），另一个是导出失败接口的名称汇总列表。这样就知道一共要导出多少数据、成功多少、哪些接口导出失败了，便于导出完成后进行数据统计。
- 第 30 行代码：打开相对目录下的一个 Excel 文件，此处位置是相对路径，即当前.py 文件目录的上一级目录和 report 目录下的指定文件的拼接路径。
- 第 31 行代码：复制已打开的 Excel 文件。
- 第 32 行代码：获取当前的系统时间戳，格式是“%Y%m%d%H%M%S”，即“年/月/日/时/分/秒”。
- 第 33 行代码：定义要生成的 Excel 文件路径，使用的是 str 类型字符串拼接的方法。
- 第 34 行代码：使用.save 方法保存新表格到 filepath 目录下。
- 第 35 行代码：判断关键参数是否在返回包数据的一级键中。如果在，则继续处理；否则直接返回第 52、53 行代码（返回 JSON 格式的 result）。
- 第 36、37 行代码：循环导出接口列表，获取每个接口在接口测试用例表中的有效数据。
- 第 39~41 行代码：打开和复制指定目录下的 Excel 文件，并对其增加工作表。工作表名称就是接口名称。
- 第 42、43 行代码：循环配置文件中的字段，在新增的 sheet 中添加标题字段，在

第 0 行的 col 列下增加参数 self.field[col]的值。

- 第 44~46 行代码：循环接口用例数据，并循环写入对应的工作表中。
- 第 48~51 行代码：如判断导出结果失败，则将失败接口名称写入失败接口列表中。

代码：analyse.py

```
 61  if __name__ == "__main__":
 62      names_export=operation_db.select_one("select value_config from config_
total where status=1 and key_config='name_export'") #获取导出的接口数据元组
 63      if names_export['code']=='0000':            #判断查询结果
 64          temp_export=eval(names_export['data'][0])#获取查询数据，并将其转换
成字典
 65          test_analyse_data=AnalyseData()
 66          result_export=test_analyse_data.export2excel(temp_export)#导出结果
 67          print (result_export)
 68      else:
 69          print('获取导出接口集失败')
```

下面是对上一段代码的解释。

- 第 62 行代码：从配置表中获取本次要导出哪些接口的列表数据。
- 第 63~69 行代码：获得测试数据，调用导出数据的方法获得对应的结果。

8.3 整体业务流程图

整体业务流程图是对前面各个零散知识点的串联。细心的读者可能已经发现了每个.py 文件的编写规律，接下来要介绍在每个独立的.py 文件运行正常的情况下，如何实现其有效地串联和被正常使用。

如果读者是按正常顺序阅读到这里的，那么应该已经掌握了以下的内容：

（1）用 Python 从数据库接口用例表中获得所有有效的测试数据。

（2）用 Python 循环这些数据，发送 HTTP 请求。

（3）用 Python 获取返回包数据，并按照测试用例表中数据的预期结果做判断。

（4）用 Python 将返回包数据比较结果写入数据库接口用例表对应的用例数据。

（5）用 Python 导出指定的接口数据到 Excel 中。

图 8-3-1 所示的是整体业务流程图。

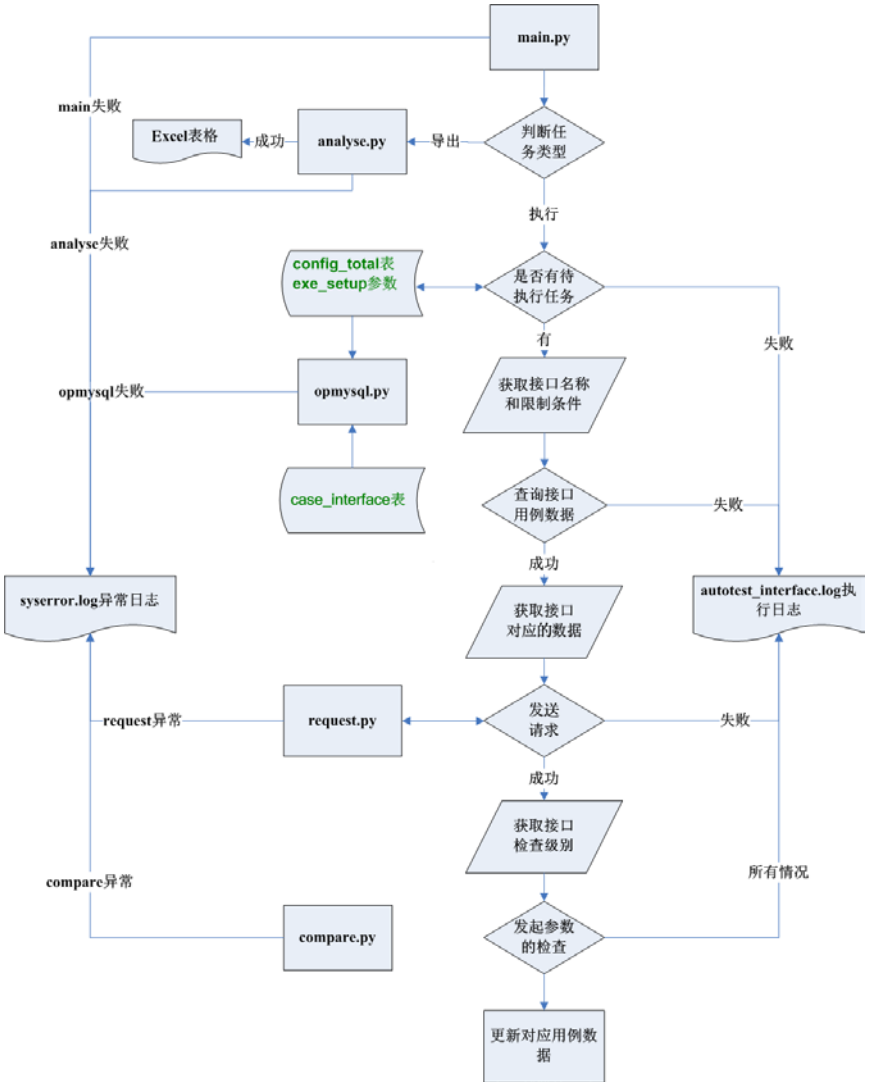


图 8-3-1　整体业务流程图

其中启动代码（即主代码 main.py）在下一章要提到的文件中，所有的触发逻辑都在其中实现，需要什么模块就调用什么模块。读者通过这个业务流程图可以大致了解各代码之间是怎样串联的。如果一时不能理解，那么没有关系，本书会在第 9 章中结合代码具体讲解。

8.4 补充知识点

8.4.1 Python 时间戳

时间戳是将正常时间格式转换成的一串数字。比如，要比较创建文件的时间先后关系，则文件的名称可以时间戳来命名。时间戳具体是如何获得的，以及如何以不同形式存在的呢?

下面的代码是将正常的时间格式转换成时间戳格式，请读者运行后理解格式的限制。

```
# -*- coding:utf-8 -*-
import datetime
dt=datetime.datetime.now()    #获取当前时间格式
p=dt.strftime("%Y%m%d%H%M%S") #将时间格式转换成指定格式的时间戳
m=dt.strftime("%Y%m%d%H")
print(dt)
print(p)
print(m)
```

8.4.2 Excel 表格的操作

Python 对于 Excel 表格的操作需要借助第三方代码库中的工具，比如 xlrd。对 Excel 表格的操作重点是表格和单元格。xlrd 模板中的基本方法见下面的代码：

```
import xlrd
from xlutils.copy import copy
workbook = xlrd.open_workbook('excel.xls')    #打开 Excel 文件
workbooknew = copy(workbook)                  #复制 Excel 文件
ws = workbooknew.get_sheet(0)                 #使用索引获取 Excel 文件的表单
ws.write(3, 0, 'changed!')                    #在对应的单元格中写入数据
workbooknew.save('newexcel.xls')              #保存 Excel 文件到指定目录下
```

9 接口自动化起航

本章讲什么：

1. 代码之外的事情；
2. 最后一个.py 文件——启动接口自动化测试。

本章是全书的倒数第 2 章。之前的内容只是零散地介绍了 HTTP 接口自动化的每个功能点。本章将这些功能点串联起来，以一个主代码来串联所有的.py 文件。

9.1 提前工作

先简单回顾一下前面部分章节所讲的知识点：

（1）第 2 章接口基础，通过该章读者可以理解什么是接口，以及接口测试的意义。

（2）第 3 章接口手工测试，通过该章读者可以了解常用的接口测试工具，掌握 Fiddler 工具的使用、接口测试用例的设计方法。

（3）第 5 章用 Python 操作 MySQL 数据库，通过该章读者可以掌握 Python 对数据库的增加、删除、修改、查询操作。

（4）第 6 章用 Python 发送 HTTP 请求，通过该章读者可以掌握 Python 调用 requests 模块发送请求，并获取返回包。

（5）第 7 章用 Python 处理 HTTP 返回包，通过该章读者可以掌握 Python 对返回包的两个层级检查。

（6）第 8 章用 Python 导出测试数据，通过该章读者可以掌握利用 Python 将数据库数据导出到 Excel 中，以及 Excel 的简单操作。

9.2 代码之外

9.2.1 初始化数据

在 config_total 表中新增导出数据设置：

```
#新增待执行的接口及其要求
INSERT INTO 'config_total' ('key_config', 'value_config', 'description', 'status')
VALUES     ('exe_setup',     '{\'getIpInfo.php\':{\'level_check\':[0,1],\
'level_exe\':[0,1,2]}}', '执行接口的条件设置，{接口名称:{检查级别:[0,1],执行级
```

```
别:[0,1,2]}}。检查级别中[0,1]代表 code 和参数完整性检查,执行级别中[0,1]分别代表 BVT,
1 级用例', '1');
```

接下来，在 config_total 表中新增一条待执行的任务数据。当然，读者也可以建立一张 task 表，并在表中存储数据。新增的这条数据有什么作用呢？主要是在运行 main.py 文件时使用。下面先来分析这个新增数据中字段的含义，分析完以后读者就会明白什么时候增加数据，以及增加何种数据了。

（1）key_config 字段，其值为 exe_setup。该字段的值是固定的，以固定的字段值作为标志（即代码获取运行任务的关键字），请读者注意。

（2）value_config 字段，其值为一个复杂的字典形式。请查看 description 字段的值。

- getIpInfo.php 是接口用例表中的接口名称，对应的字段是 name_interface。
- level_check 键的值是[0,1]，即 nodes 这个接口所要做的是关键参数值检查和参数完整性检查。如只想做某一项检查，则在列表中只需配置某一项的值。比如，只检查关键参数值，则只需要配置[0]。
- level_exe 键的值是[0,1,2]。大家知道，每个测试用例都有级别，这里就是用来控制代码要执行哪些级别的用例，[0,1,2]代表的是 BVT、1 级、2 级（读者也可以自行设定），其对应的是 case_interface 表中的 exe_level 字段的值。

（3）status 代表当前任务的状态。

9.2.2 代码结构图

图 9-2-1 是完整的 HTTP 接口自动化测试的代码结构。如果读者阅读了之前的章节，那么对于这个结构不会陌生，而且我相信你现在的目录结构应该和这个是相同的。当然，这个结构不是固定的，你可以根据自己的实际需要做相应的调整。总的原则是结构清晰，便于其他人能很好地理解代码结构。

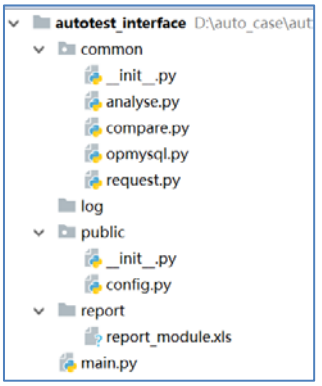


图 9-2-1　代码的完整结构

9.3　接口自动化起航代码

9.3.1　业务逻辑梳理

8.3 节有一张整体业务流程图（见图 8-3-1）。细心的读者可能已经理解了整个业务的逻辑，此图也是本章所有代码实现的逻辑。下面对该图的业务逻辑进行分析：

（1）启动 main.py 代码。可以将 main.py 理解为一个可执行程序。

（2）等待用户手动输入值，判断用户输入值是否正确。

（3）对于正确的值再判断任务类型是导出数据还是执行用例。

- 如果用户选择导出数据，则直接调用 analyse.py 代码导出对应的测试用例数据。
- 如果用户选择执行用例，则代码判断是否存在有效的待执行任务。

（4）如果存在有效的待执行任务，则调用 opmysql.py 从 config_total 表中获取该接口的名称和限制条件。如果没有，则在控制台输出提示信息。

（5）依据接口名称和限制条件调用 opmysql.py 从 case_interface 表查询接口用例数据，如果查询失败，则在控制台输出提示信息，如果查询成功，则继续下一步。

（6）获取接口对应的用例数据。

（7）调用 request.py 发送 HTTP 请求，失败则在控制台输出提示信息，成功继续下一步。

（8）循环检查级别数据。

（9）依据检查级别数据调用 compare.py，对返回包数据做处理，如果失败，则在控制台输出提示信息，如果成功，则继续下一步。

（10）更新接口用例表所对应的数据。

9.3.2　代码实例

1．新建 main.py 文件

在工程目录下新建一个 main.py 文件。当然，也可以按照自己的习惯来命名，具体步骤如下所示。

（1）在工程文件夹上单击鼠标右键，在弹出的快捷菜单中选择“New”→“Python file”命令，新建.py 文件，将“Name”设为“main”，单击“OK”按钮，如图 9-3-1 所示。

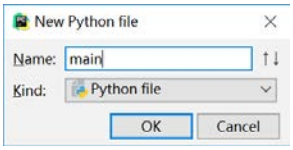


图 9-3-1　新建 main.py 文件

（2）新建工程及其目录下的.py 文件。

2．在.py 文件下创建代码

代码：main.py

```
1 # -*- coding:utf-8 -*-
2 #__author__ = '大婶 N72'
3 '''
4 主文件，用于串联各个.py 文件
5 '''
6 import logging,os,re
```

```
 7  from common import request, opmysql, analyse, compare
 8  from public import config
 9  base_request= request.RequestInterface()#实例化 HTTP 请求
 10  base_operationdb_interface= opmysql.OperationDbInterface()#实例化接口测
试数据库操作类
 11  try:
 12      print("开始接口自动化程序，请选择操作类型（0|执行用例；1|导出测试结果)")
 13      value_input = input('请输入操作类型:')#该处输入 0 时，调试窗口是 str，而可
视化窗口是 int
 14      while not re.search(r'^[0-1]$',value_input):#判断不是 0 和 1 的数字
 15          print ("请输入正确的操作类型(0|执行用例；1|导出测试结果)")
 16          value_input = str(input('请输入操作类型:'))
 17      else:
 18          if value_input=='0':
 19              print("您输入的是：0|执行测试用例")
 20                    module_execute=base_operationdb_interface.select_all
("SELECT value_config from config_total where key_config='exe_setup' and
status=1")#获取待执行接口数据
 21                          if len(module_execute['data'])!=0 and module_
execute ['code']=='0000':
 22                for module_execute_one in module_execute['data']:
 23                              temp_module_execute=eval(module_execute_one
['value_config'])#每个接口的字典数据
 24                            for temp_name_interface,condition in
temp_module_execute.items():
 25                            print('############开始执行接口：%s############\
n' %(temp_name_interface))
 26                        temp_level_check=condition['level_check']#检查级别
 27                         temp_level_exe=tuple(condition['level_exe'])#执行
级别
 28                        data_case_interface=base_operationdb_interface.
select_all("select * from case_interface where case_status=1 and
name_interface='%s' and exe_level in %s" %
(temp_name_interface,temp_level_exe))#获取接口测试数据
 29                        if data_case_interface['code']=='0000' and
len(data_case_interface['data'])!=0:
 30                            for temp_case_interface in
data_case_interface['data']:
 31                            id_case=str(temp_case_interface['id'])#用例编号
```

```
 32                                      url_interface=temp_case_interface
['url_interface']#接口地址
 33
headerdata=eval(temp_case_interface['header_interface'])#请求头文件
 34
param_interface=temp_case_interface['params_interface']#接口请求参数
 35
type_interface=temp_case_interface['exe_mode']#执行环境
 36
result_http_response=base_request.http_request
(interface_url=url_interface,headerdata=headerdata,interface_param=param_
interface,request_type=type_interface)#发送 HTTP 请求
 37                                  print("接口地址:%s\n 请求参数:%s\n 返回包数
据:%s" %(url_interface,param_interface,result_http_response))
 38                                base_operationdb_interface.op_sql("UPDATE
case_interface          set          result_interface='%s'          where
id=%s" %(result_http_response['data'],id_case))#将接口返回包写入用例表
 39                               if result_http_response['code']=='0000' and
len(result_http_response['data'])!=0:
 40                                      for child_level_check in
temp_level_check:#循环检查级别
 41                                      base_compare=
compare.CompareParam(temp_case_interface)#实例化参数比较类
 42                                      if child_level_check in (0,):#执行关键
参数值检查
 43
result_compare_code=base_compare.compare_code(result_http_response['data'
])
 44                                      print('用例编号：%s|检查级别：关键参数值|
接口名称：%s|提示信息：%s\n' %(id_case,temp_name_interface,result_
compare_code['message']))
 45                                      elif child_level_check in [1]:#执行参
数完整性检查
 46
result_compare_params_complete=base_compare.compare_params_complete(resul
t_http_response['data'])
 47                                       print('用例编号：%s|检查级别：参数完整性|
接口名称：%s|提示信息：%s\n' %(id_case,temp_name_interface,
result_compare_params_complete['message']))
```

```
48                          elif child_level_check in [2]:#执行功能测试，待开发
49                                      pass
50                          elif child_level_check in (3,):#执行结构完整性检查，
待开发
51                                      pass
52                          else:
53                          print('用例编号：%s|接口名称：%s|检查级别错
误：%s\n' %(id_case,temp_name_interface,child_level_check))
54                                  elif len(result_http_response['data'])==0:
55                                      print('用例编号：%s|接口名称：%s|错误信息：接
口返回数据为空\n' %(id_case,temp_name_interface))
56                                  else:
57                                          print('用例编号：%s|接口名称：%s|错误信
息：%s\n' %(id_case,temp_name_interface,result_http_response['message']))
58                          elif len(data_case_interface['data'])==0:
59                              print('接口名称：%s|错误信息：获取用例数据为空，请检
查用例\n' %(temp_name_interface))
60                          else:
61                              print('接口名称：%s|错误信息：获取用例数据失败|错误信
息：%s\n' %(temp_name_interface,data_case_interface['message']))
62                                  print('#############结束执行接
口： %s#############\n' %(temp_name_interface))
63              else:
64                              print('错误信息：待执行接口获取失败|错误信
息：%s' %module_execute['message'])
65          elif value_input=='1':
66                      print('您输入的是：1|导出测试结果，请注意查看目
录：%s' %(config.src_path+'\\report'))
67              names_export=base_operationdb_interface.select_one("select
value_config from config_total where status=1 and key_config='name_export'")#
获取导出的接口数据元组
68                              if names_export['code']=='0000' and
len(names_export['data']['value_config'])!=0:#判断查询结果
69                  temp_export=eval(names_export['data']['value_config'])#
获取查询数据，并将其转换成字典
70                  test_analyse_data= analyse.AnalyseData()#实例化数据分析类
71                  result_export=test_analyse_data.export2excel
(temp_export)#导出结果
72                  print (result_export['message'])
```

```
 73                print ("导出失败接口列表：%s\n" %result_export['data'])
 74            else:
 75                print("请检查配置表数据正确性，当前值
为：%s\n" %names_export['data'])
 76  except Exception as error:#记录日志到 log.txt 文件
 77      print("系统出现异常：%s" %error)
 78                logging.basicConfig(filename   =     config.src_path   +
'\log\syserror.log', level = logging.DEBUG,format='%(asctime)s %(filename)
s[line:%(lineno)d] %(levelname)s %(message)s')
 79      logger = logging.getLogger(__name__)
 80      logger.exception(error)
 81  input('Press Enter to exit...')
```

下面是对上一段代码的解释。

- 第 7 行代码：从 common 包中导出需要的几个模块——请求模块 request、数据库操作模块 opmysql、结果导出模块 analyse、返回包处理模块 compare。
- 第 9、10 行代码：分别实例化 HTTP 请求类和数据库操作类。
- 第 12 行代码：由于要把 main.py 打造成可执行的程序，则可双击该.py 文件直接执行。
- 第 13 行代码：使用 input 方法捕获控制台的输入信息，并将其强制转换成整型数值。注意，此处笔者对输出的中文信息做了两次转码，所以建议读者能用英文就别用中文。
- 第 14~17 行代码：对捕获的参数做判断。如果输入的数不是 0 或 1，则持续让用户输入。如果是 0 或 1，则继续下面的逻辑。
- 第 18 行代码：如用户输入的是 0，则执行配置文件中设置的接口测试用例。
- 第 20 行代码：调用数据库操作实例下的 select_all 方法以获取待执行接口数据。这个数据可能不止一条，因为按照本章前面的初始化数据，每个接口会初始化一条待执行的任务，所以获取的可能是多条数据组成的列表。
- 第 21~23 行代码：判断查询数据成功且存在待执行的任务（列表不为空），循环获取每个接口限制条件数据，用于获取单个接口的具体限制条件。
- 第 24 行代码：利用字典的 items()方法获取接口具体限制条件的键值对元组。

- 第 28 行代码：获取接口在接口用例表中的测试数据。
- 第 36 行代码：调用 request.py 中发送 HTTP 请求的方法处理接口请求，获得返回值，将其赋值给 result_http_response。
- 第 37 行代码：用 print 格式化输出 HTTP 请求过程参数，便于直接分析定位。
- 第 38 行代码：将接口返回包数据更新到对应的接口用例中。
- 第 39、40 行代码：判断返回包的数据正确性以及返回数据不能为空，循环接口待检查级别列表。
- 第 41 行代码：实例化参数比较的类，用于后面返回包的数据处理。
- 第 42~53 行代码：按照实际的检查级别，按条件执行对应的检查方法，这里保留了两个功能和结构完整的检查方法为后期开发使用，处理完成之后用 print 格式化输出结果。
- 第 65 行代码：如果用户输入的是 1，则导出测试数据到 Excel 中。
- 第 67 行代码：获取数据库 config_total 表中待导出的接口列表。
- 第 68~71 行代码：如果判断待导出的数据列表正确，则调用导出方法。其中第 69 行代码为获取待导出的数据列表。第 71 行代码为调用导出数据的方法，获得导出结果并赋值给 result_export。
- 第 72、73 行代码：分别打印出导出结果信息及导出失败的接口列表。
- 第 81 行代码：退出可执行程序命令。

9.4 代码操作步骤

（1）启动代码文件：双击 main.py 文件，启动 main 窗口，如图 9-4-1 所示。用户可输入对应的数字（0：执行测试用例；1：导出测试结果）。

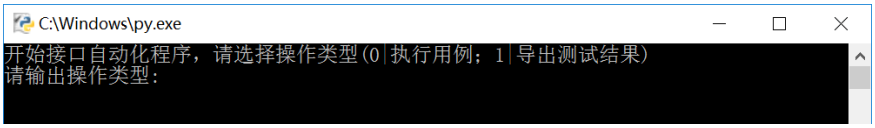


图 9-4-1　启动 main 窗口

（2）图 9-4-2 所示的是选择 0 时的窗口。

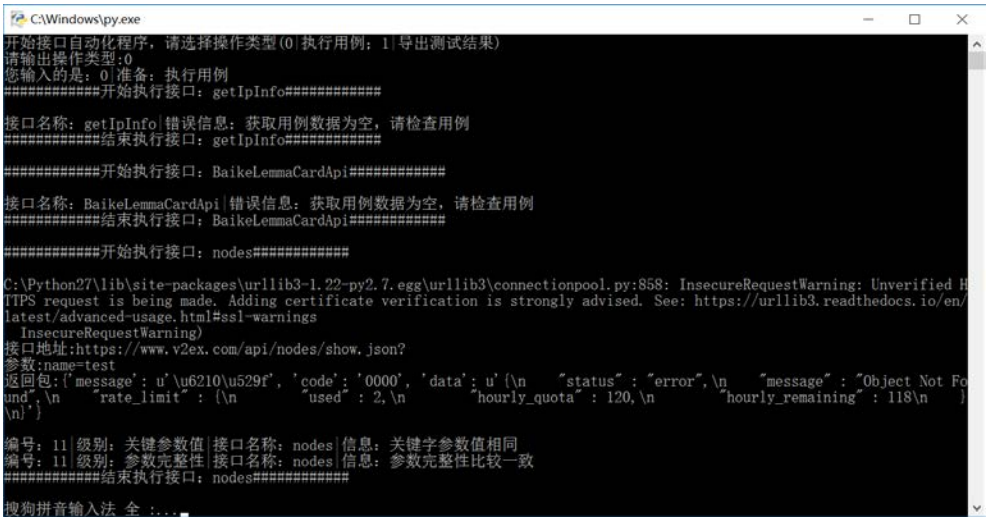


图 9-4-2　执行测试用例窗口

（3）选择 1，导出测试结果，如图 9-4-3 所示。

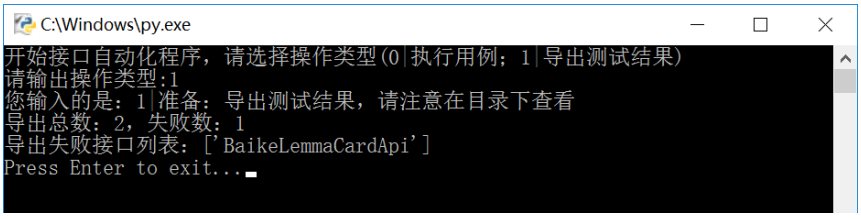


图 9-4-3　导出测试结果

（4）导出测试报告，如图 9-4-4 所示。

图 9-4-4　导出测试报告

9.5 补充知识点

9.5.1 用 print 格式化输出

Python 中内置的%操作符和 format 函数，都可用于格式化字符串。

1．%操作符的使用

- %o：oct，八进制。
- %d：dec，十进制。
- %x：hex，十六进制。
- %f：保留小数点后面 6 位有效数字。如果是%.3f，则保留 3 位小数位。
- %e：保留小数点后面 6 位有效数字，按指数形式输出。如果是%.3e，则保留 3 位小数位，使用科学记数法。
- %g：如果有 6 位有效数字，则使用小数方式，否则使用科学记数法。如果是%.3g，则保留 3 位有效数字，使用小数方式或科学记数法。
- %s：字符串。
- %10s：右对齐，占位符 10 位。
- %-10s：左对齐，占位符 10 位。
- %.2s：截取两位字符串。
- %10.2s：10 位占位符，截取两位字符串。
- \：续行符。
- \\：反斜杠。
- \'：单引号。
- \"：双引号。

举例如下：

```
print('His name is %s' % 'Aviad')
print("He is %d years old" %25)
print("His height is %f m" %1.83)
print('%10.2s' % 'hello world')
```

2．format()方法

format()方法功能更强大。该方法把字符串当成一个模板，通过传入的参数进行格式化，并且使用大括号“{}”作为特殊字符代替“%”。

使用方法有两种格式：b.format(a)和 format(a,b)。

（1）不带编号，即“{}”。

（2）带数字编号，可调换顺序，即“{1}”“{2}”。

（3）带关键字，即“{a}”“{tom}”。

```
print('{} {}'.format('hello','world'))                  # 不带字段
print('{0} {1}'.format('hello','world'))                # 带数字编号
print('{0} {1} {0}'.format('hello','world'))            # 打乱顺序
print('{a} {tom} {a}'.format(tom='hello',a='world'))    # 带关键字
```

9.5.2 数据驱动和关键字驱动

数据驱动（data-driven testing，DDT）和关键字驱动（keyword-driven testing，KDT）是有区别的：什么因素对测试结果起决定性作用，就是什么因素驱动测试过程。

（1）驱动因素：KDT 是 action 和 data，DDT 是 data。

（2）执行过程关注：KDT 需要定义执行过程（白盒），而 DDT 不关注（黑盒）。

（3）精细度：KDT 是动作级的，DDT 是函数级的。

（4）KDT 是 DDT 的低级版本（越靠近实际操作过程，越低级。类似于编程语言，越接近机器码，越低级）。

（5）KDT 完成对 action 的定义封装之后，趋近 DDT。

（6）KDT 使用面向对象思想将测试过程抽象为 action word，可以定义用户的每个 action。

10 实际接口场景演示

本章讲什么：

通过在线可使用的接口来演示接口自动化测试过程。

10.1 提前工作

在开始本章的接口操练之前，请读者先确定已经掌握以下知识点：

（1）设计接口测试用例的方法。

（2）搭建自动化代码的运行环境。

（3）启动程序和终止程序。

10.2 接口举例

可以使用第三方提供的接口来做测试，比如淘宝提供的根据 IP 地址获取信息的接口：

http://ip.taobao.com/service/getIpInfo.php?ip=63.223.108.4

10.3 准备与执行

10.3.1 设计接口测试用例

依据对接口的理解，使用思维导图工具设计接口测试用例框架，梳理要测试的点及其参数类型，详见图 10-3-1。

10.3.2 按照接口用例设计准备测试数据

使用工具设计完成测试用例框架，并定义好其中的参数场景，然后将具体数据加入数据库 case 表中作为完整的测试数据，如图 10-3-2 所示。

10.3.3 在 config_total 表中增加执行与导出配置项

在准备完测试数据之后，配置好执行所需要的一些参数并导出配置表，如图 10-3-3 所示。

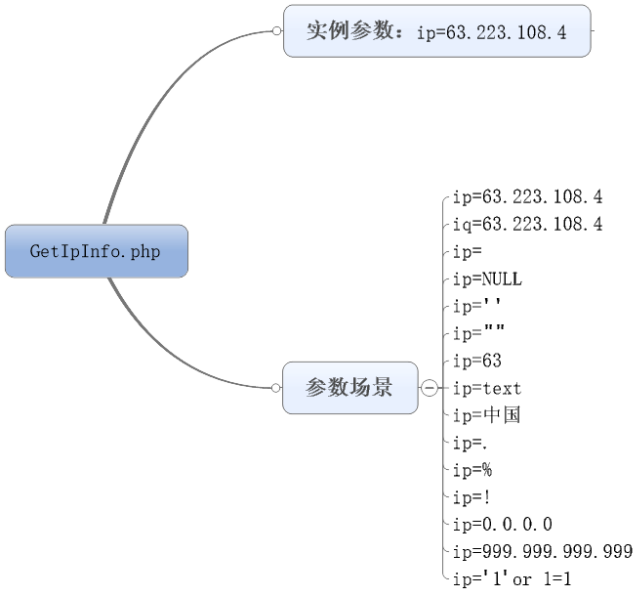


图 10-3-1　接口用例设计

id	name_interface	exe_level	exe_mode	url_interface	header_interface	params_interface	result_interface	code_to_compa
1	getIpInfo.php	0	GET	http://ip.taobao.com/service/getIpInfo.php?	{'Host':'ip.taobao.com'}	ip=63.223.108.4		code
2	getIpInfo.php	0	GET	http://ip.taobao.com/service/getIpInfo.php?	{'Host':'ip.taobao.com'}	iq=63.223.108.4		code
3	getIpInfo.php	0	GET	http://ip.taobao.com/service/getIpInfo.php?	{'Host':'ip.taobao.com'}	ip=		code
4	getIpInfo.php	0	GET	http://ip.taobao.com/service/getIpInfo.php?	{'Host':'ip.taobao.com'}	ip=NULL		code
5	getIpInfo.php	0	GET	http://ip.taobao.com/service/getIpInfo.php?	{'Host':'ip.taobao.com'}	ip=''		code
6	getIpInfo.php	0	GET	http://ip.taobao.com/service/getIpInfo.php?	{'Host':'ip.taobao.com'}	ip=""		code
7	getIpInfo.php	0	GET	http://ip.taobao.com/service/getIpInfo.php?	{'Host':'ip.taobao.com'}	ip=63		code
8	getIpInfo.php	0	GET	http://ip.taobao.com/service/getIpInfo.php?	{'Host':'ip.taobao.com'}	ip=text		code
9	getIpInfo.php	0	GET	http://ip.taobao.com/service/getIpInfo.php?	{'Host':'ip.taobao.com'}	ip=中国		code
10	getIpInfo.php	0	GET	http://ip.taobao.com/service/getIpInfo.php?	{'Host':'ip.taobao.com'}	ip=.		code
11	getIpInfo.php	0	GET	http://ip.taobao.com/service/getIpInfo.php?	{'Host':'ip.taobao.com'}	ip=%		code
12	getIpInfo.php	0	GET	http://ip.taobao.com/service/getIpInfo.php?	{'Host':'ip.taobao.com'}	ip=!		code
13	getIpInfo.php	0	GET	http://ip.taobao.com/service/getIpInfo.php?	{'Host':'ip.taobao.com'}	ip=0.0.0.0		code
14	getIpInfo.php	0	GET	http://ip.taobao.com/service/getIpInfo.php?	{'Host':'ip.taobao.com'}	ip=999.999.999.999	(Null)	code
15	getIpInfo.php	0	GET	http://ip.taobao.com/service/getIpInfo.php?	{'Host':'ip.taobao.com'}	ip='1'or '1'='1'		code

图 10-3-2　接口测试数据

id	key_config	value_config	description
1	name_export	['getIpInfo.php']	导出接口数据配置
2	exe_setup	{'getIpInfo.php':{'level_check':[0,1],'level_exe':[0,1,2]}}	执行接口的条件设置，(接口名称:(检查

图 10-3-3　导出配置表

10.3.4　执行 main.py

（1）选择 0，执行测试用例，如图 10-3-4 所示。

图 10-3-4　main.py 执行界面

（2）选择 1，导出执行的结果，如图 10-3-5 所示。

编号	接口名称	用例级别	请求类型	接口地址	接口头文件	接口请求参	接口返回包	待比较参数	实际参数值	预期参数值	参数值比较	待比较参数集合	实际参数集合	参数完整
1	getIpInfo.p	0	GET	http://ip.ta	{'Host':'ip.t	ip=63.223.	{"code":0,"data":{"ip":"63	code	0	0	1	['code','data','cc	['region_id', 'cou	1
2	getIpInfo.p	0	GET	http://ip.ta	{'Host':'ip.t	iq=63.223.	[]	code		0	None	['code','data','country']	None	
3	getIpInfo.p	0	GET	http://ip.ta	{'Host':'ip.t	ip=	{"code":0,"data":{"ip":"","c	code	0	0	1	['code','data','cc	['region_id', 'cou	1
4	getIpInfo.p	0	GET	http://ip.ta	{'Host':'ip.t	ip=NULL	{"code":0,"data":{"ip":"NU	code	0	0	1	['code','data','cc	['region_id', 'cou	1
5	getIpInfo.p	0	GET	http://ip.ta	{'Host':'ip.t	ip=''	{"code":0,"data":{"ip":"&#	code	0	0	1	['code','data','cc	['region_id', 'cou	1
6	getIpInfo.p	0	GET	http://ip.ta	{'Host':'ip.t	ip=""	{"code":0,"data":{"ip":"&#	code	0	0	1	['code','data','cc	['region_id', 'cou	1
7	getIpInfo.p	0	GET	http://ip.ta	{'Host':'ip.t	ip=63	{"code":0,"data":{"ip":"63	code	0	0	1	['code','data','cc	['region_id', 'cou	1
8	getIpInfo.p	0	GET	http://ip.ta	{'Host':'ip.t	ip=text	{"code":0,"data":{"ip":"te	code	0	0	1	['code','data','cc	['region_id', 'cou	1
9	getIpInfo.p	0	GET	http://ip.ta	{'Host':'ip.t	ip=中国	{"code":0,"data":{"ip":"&#	code	0	0	1	['code','data','cc	['region_id', 'cou	1
10	getIpInfo.p	0	GET	http://ip.ta	{'Host':'ip.t	ip=.	{"code":0,"data":{"ip":".",	code	0	0	1	['code','data','cc	['region_id', 'cou	1
11	getIpInfo.p	0	GET	http://ip.ta	{'Host':'ip.t	ip=%	[]	code		0	None	['code','data','country']	None	
12	getIpInfo.p	0	GET	http://ip.ta	{'Host':'ip.t	ip=!	{"code":0,"data":{"ip":"!",	code	0	0	1	['code','data','cc	['region_id', 'cou	1
13	getIpInfo.p	0	GET	http://ip.ta	{'Host':'ip.t	ip=0.0.0.0	{"code":0,"data":{"ip":"0.	code	0	0	1	['code','data','cc	['region_id', 'cou	1
14	getIpInfo.p	0	GET	http://ip.ta	{'Host':'ip.t	ip=999.99	{"code":0,"data":{"ip":"99	code	0	0	1	['code','data','cc	['region_id', 'cou	1
15	getIpInfo.p	0	GET	http://ip.ta	{'Host':'ip.t	ip='1'or '1'	{"code":0,"data":{"ip":"&#	code	0	0	1	['code','data','cc	['region_id', 'cou	1

图 10-3-5　导出执行的结果

附录A 本书用到的 Python 代码清单

代码：opmysql.py

```
# -*- coding:utf-8 -*-
#_ _author_ _ = '大婶 N72'
'''
定义对 MySQL 数据库基本操作的封装
1.包括基本的单条语句操作，删除、修改、更新
2.独立地查询单条、多条数据
3.独立地添加多条数据
'''
import logging,os,MySQLdb
from public import config
class OperationDbInterface(object):
    def _ _init_ _(self,host_db='192.168.0.106',user_db='root',passwd_db=
'root',name_db='test_interface',port_db=3306,link_type=0):
        '''
        :param host_db: 数据库服务主机
        :param user_db: 数据库用户名
        :param passwd_db: 数据库密码
```

```
        :param name_db: 数据库名称
        :param port_db: 端口号，整型数字
        :param link_type: 连接类型，用于输出的数据是元组还是字典，默认是字典，
link_type=0
        :return:游标
        '''
        try:
            self.conn=MySQLdb.connect(host=host_db,user=user_db,
passwd=passwd_db, db=name_db, port=port_db, charset='utf8')#创建数据库链接
            if link_type==0:
                self.cur=self.conn.cursor(cursorclass                    =
pymysql.cursors.DictCursor)#返回字典
            else:
                self.cur=self.conn.cursor()#返回元组
        except pymysql.Error as e:
            print("创建数据库连接失败|Mysql Error %d: %s" % (e.args[0],
e.args[1]))
            logging.basicConfig(filename = config.src_path + '/log/syserror.
log',   level   =   logging.DEBUG,   format='%(asctime)s   %(filename)s
[line:%(lineno)d] %(levelname)s %(message)s')
            logger = logging.getLogger(_ _name_ _)
            logger.exception(e)
    #定义单条数据操作，增加、删除、修改
    def op_sql(self,condition):
        '''
        :param condition: SQL 语句，该通用方法可用来替代 insertone、updateone、
deleteone
        :return:字典形式
        '''
        try:
            self.cur.execute(condition)#执行 SQL 语句
            self.conn.commit()#提交游标数据
            result={'code':'0000','message':'执行通用操作成功','data':[]}
        except pymysql.Error as e:
            self.conn.rollback()  # 执行回滚操作
            result={'code':'9999','message':'执行通用操作异常','data':[]}
            print("数据库错误|op_sql %d: %s" % (e.args[0], e.args[1]))
            logging.basicConfig(filename     =     config.src_path     +
'/log/syserror.log',   level   =   logging.DEBUG,   format='%(asctime)s%
```

```
(filename)s[line:%(lineno)d] %(levelname)s %(message)s')
            logger = logging.getLogger(_ _name_ _)
            logger.exception(e)
        return result

        # 查询表中单条数据
    def select_one(self, condition):
        '''
        :param condition: sql 语句
        :return:字典形式的单条查询结果
        '''
        try:
            rows_affect = self.cur.execute(condition)
        if rows_affect > 0:  # 查询结果返回数据数大于 0
            results = self.cur.fetchone()  # 获取一条结果
            result = {'code': '0000', 'message': u'执行单条查询操作成功', 'data':
results}
            else:
                result = {'code': '0000', 'message': u'执行单条查询操作成功',
'data': []}
        except pymysql.Error as e:
            self.conn.rollback()  # 执行回滚操作
            result = {'code': '9999', 'message': u'执行单条查询异常', 'data': []}
            print("数据库错误|select_one %d: %s" % (e.args[0], e.args[1]))
            logging.basicConfig(filename      =      config.src_path      +
'/log/syserror.log',                                  level=logging.DEBUG,
format='%(asctime)s %(filename)s[line:%(lineno)d] %(levelname)s %(message
)s')
            logger = logging.getLogger(__name__)
            logger.exception(e)
        return result
    #查询表中多条数据
    def select_all(self,condition):
        '''
        :param condition: SQL 语句
        :return:字典形式的批量查询结果
        '''
        try:
            rows_affect=self.cur.execute(condition)
```

```
            if rows_affect>0:#查询结果返回数据数大于 0
                self.cur.scroll(0, mode='absolute') # 鼠标光标回到初始位置
                results = self.cur.fetchall()#返回游标中所有结果
                result={'code':'0000','message':' 执 行 批 量 查 询 操 作 成 功
','data':results}
            else:
                result={'code':'0000','message':' 执 行 批 量 查 询 操 作 成 功
','data':[]}
        except pymysql.Error as e:
            self.conn.rollback()  # 执行回滚操作
            result={'code':'9999','message':'执行批量查询异常','data':[]}
            print("数据库错误|select_all %d: %s" % (e.args[0], e.args[1]))
            logging.basicConfig(filename      =      config.src_path      +
'/log/syserror.log',           level           =           logging.DEBUG,
format='%(asctime)s %(filename)s[line:%(lineno)d] %(levelname)s %(message
)s')
            logger = logging.getLogger(_ _name_ _)
            logger.exception(e)
        return result
    #定义表中插入多条数据操作
    def insert_more(self,condition,params):
        '''
        :param condition: insert 语句
        :param params: insert 数据，列表形式[('3','Tom','1 year 1 class','
6'),('3','Jack','2 year 1 class','7'),]
        :return:字典形式的批量插入数据结果
        '''
        try:
            results=self.cur.executemany(condition,params)#返回插入的数据条数
            self.conn.commit()
            result={'code':'0000','message':'执行批量查询操作成功','data':int
(results)}
        except pymysql.Error as e:
            self.conn.rollback()  # 执行回滚操作
            result={'code':'9999','message':'执行批量查询异常','data':[]}
            print("数据库错误|insert_more %d: %s" % (e.args[0], e.args[1]))
            logging.basicConfig(filename   =   config.src_path   +   '/log/
syserror.log', level = logging.DEBUG, format='%(asctime)s %(filename)s[line:
% (lineno)d] %(levelname)s %(message)s')
```

```
            logger = logging.getLogger(_ _name_ _)
            logger.exception(e)
        return result
    #数据库关闭
    def _ _del_ _(self):
        if self.cur != None:
            self.cur.close()
        if self.conn != None:
            self.conn.close()
if _ _name_ _ == "_ _main_ _":
    test=OperationDbInterface()#实例化类
    result=test.select_all("SELECT * FROM config WHERE id=1")
    #result=test.insert_more("insert     into     config_total(key_config,
value_config,status) values(%s,%s,%s)",[(1,1,1),(2,2,2)])
    if result['code']=='0000':
        print(result['data'])
    else:
        print(result['message'])
```

代码：request.py

```
# -*- coding:utf-8 -*-
#_ _author_ _ = '大婶 N72'
'''
定义对 HTTP 请求操作的封装
1.http_request 是主方法，直接供外部调用
2._ _http_get，_ _http_post 是实际底层分类调用的方法
'''
import requests,os,logging
from common import opmysql
from public import config
class RequestInterface(object):
    #定义处理不同类型的求参数，包含字典、字符串、空值
    def _ _new_param(self,param):
        try:
            if isinstance(param,str) and param.startswith('{'):
                new_param=eval(param)
            elif param==None:
                new_param=''
            else:
```

```
                new_param=param
        except Exception as error:#记录日志到 log.txt 文件
            new_param=''
            logging.basicConfig(filename = config.src_path + '/log/syserror.
log',level    =    logging.DEBUG,format='%(asctime)s    %(filename)s[line:%
(lineno)d] %(levelname)s %(message)s')
            logger = logging.getLogger(_ _name_ _)
            logger.exception(error)
        return new_param
    # POST 请求，参数在 body 中
    def _ _http_post(self,interface_url,headerdata,interface_param):
        '''
        :param interface_url: 接口地址
        :param headerdata: 请求头文件
        :param interface_param: 接口请求参数
        :return:字典形式结果
        '''
        try:
            if interface_url!='':
                temp_interface_param = self._ _new_param(interface_param)
                response = requests.post(url=interface_url,
headers=headerdata,data=temp_interface_param,verify=False,timeout=10)
                if response.status_code==200:
                    durtime = (response.elapsed.microseconds) / 1000  # 发起请
求和响应到达的时间，单位 ms
                    result={'code':'0000','message':' 成 功 ','data':response.
text}
                else:
                    result={'code':'2004','message':'接口返回状态错误','data':
[]}
            elif interface_url == '':
                result={'code':'2002','message':'接口地址参数为空','data':[]}
            else:
                result = {'code': '2003', 'message': '接口地址错误', 'data': []}
        except Exception as error:#记录日志到 log.txt 文件
            result={'code':'9999','message':'系统异常','data':[]}
            logging.basicConfig(filename      =      config.src_path      +
'/log/syserror.log',level = logging.DEBUG,format='%(asctime)s %(filename)s
[line:%(lineno)d] %(levelname)s %(message)s')
```

```
            logger = logging.getLogger(_ _name_ _)
            logger.exception(error)
        return result
    # GET 请求，参数在接口地址后面
    def _ _http_get(self,interface_url,headerdata,interface_param):
        '''
        :param interface_url: 接口地址
        :param headerdata: 请求头文件
        :param interface_param: 接口请求参数
        :return:字典形式结果
        '''
        try:
            if interface_url != '':
                temp_interface_param = self._ _new_param(interface_param)
                requrl = interface_url+temp_interface_param
                response  =  requests.get(url=requrl,  headers=headerdata,
verify=False,timeout=10)
                #print response
                if response.status_code==200:
                    durtime = (response.elapsed.microseconds) / 1000  # 发起请
求和响应到达的时间，单位为 ms
                    result={'code':'0000','message':' 成 功 ','data':response.
text}
                else:
                     result={'code':'3004','message':'接口返回状态错误','data':
[]}
            elif interface_url == '':
                result={'code':'3002','message':'接口地址参数为空','data':[]}
            else:
                result={'code':'3003','message':'接口地址错误','data':[]}
        except Exception as error:#记录日志到 log.txt 文件
            result={'code':'9999','message':'系统异常','data':[]}
            logging.basicConfig(filename = config.src_path + '/log/syserror.
log',level                                                                  =
logging.DEBUG,format='%(asctime)s %(filename)s[line:%(lineno)d] %(levelna
me)s %(message)s')
            logger = logging.getLogger(_ _name_ _)
            logger.exception(error)
        return result
```

```
    # 统一处理 HTTP 请求
    def       http_request(self,interface_url,headerdata,interface_param,
request_type):
        '''
        :param interface_url: 接口地址
        :param headerdata: 请求头文件
        :param interface_param: 接口请求参数
        :param request_type:请求类型
        :return:字典形式结果
        '''
        try:
           if request_type=='get' or request_type=='GET':
              result=self._ _http_get(interface_url,headerdata,interface_param)
           elif request_type=='post' or request_type=='POST':
              result=self._ _http_post(interface_url,headerdata,interface_param)
         else:
              result={'code':'1000','message':' 请 求 类 型 错 误 ','data':
request_type}
        except Exception as error:#记录日志到 log.txt 文件
           result={'code':'9999','message':'系统异常','data':[]}
           logging.basicConfig(filename = config.src_path +
'/log/syserror.log',level = logging.DEBUG,format='% (asctime) s % (filename)
s[line:%(lineno)d] %(levelname)s %(message)s')
           logger = logging.getLogger(_ _name_ _)
           logger.exception(error)
        return result
 if _ _name_ _ == "_ _main_ _":
    test_interface=RequestInterface()#实例化 HTTP 请求类

test_db=opmysql.OperationDbInterface(host_db='192.168.1.110',user_db='roo
t',passwd_db='root',name_db='test_interface',port_db=3306,link_type=0)# 实
例化 MySQL 处理类
    sen_sql="select
exe_mode,url_interface,header_interface,params_interface from
case_interface where name_interface='getIpInfo' and  id=1"
    params_interface=test_db.select_one(sen_sql)
    if params_interface['code']=='0000':
       url_interface=params_interface['data']['url_interface']
       temp=params_interface['data']['header_interface']
```

```
        print(temp)

headerdata=eval(params_interface['data']['header_interface'])#unicode 转换
成字典
        param_interface=params_interface['data']['params_interface']
        type_interface=params_interface['data']['exe_mode']
        if url_interface!='' and headerdata!='' and param_interface!='' and
type_interface!='':
            result=test_interface.http_request(interface_url=url_interface,
headerdata=headerdata,interface_param=param_interface,request_type=type_i
nterface)
            if result['code']=='0000':
                result_resp=result['data']
                print("处理 http 请求成功，返回数据是：%s" %result_resp)
            else:
                print("处理 http 请求失败")
        else:
            print("测试用例数据中有空值")
    else:
        print("获取接口测试用例数据失败")
```

代码：compare.py

```
# -*- coding:utf-8 -*-
#_ _author_ _ = '大婶 N72'
'''
定义对 HTTP 返回包数据过程
1.compare_param 是对外的参数比较类
2.compare_code 是关键参数值比较方法，compare_params_complete 是参数完整性比较方法
3.get_compare_params 是获得返回包数据去重后集合方法
4.recur_params 递归操作方法，辅助去重使用
'''
import sys
reload(sys)
sys.setdefaultencoding('utf-8')
import json,opmysql,os,logging
from public import config
operation_db=opmysql.OperationDbInterface()#实例化测试数据库操作类
```

```
class CompareParam(object):
    #初始化数据
    def _ _init_ _(self,params_interface):
        self.params_interface=params_interface#接口入参
        self.id_case=params_interface['id']#测试用 id
        self.result_list_response=[]#定义用来存储参数集的空列表
        self.params_to_compare=params_interface['params_to_compare']#定义参
数完整性预期结果
    #定义关键参数值(code)比较
    def compare_code(self,result_interface):
        '''
        :param result_interface: HTTP 返回包数据
        :return:返回码 code，返回信息 message，数据 data
        '''
        try:
            if result_interface.startswith('{') and isinstance
(result_interface,str):
                temp_result_interface=json.loads(result_interface)#将字符类型
转换成字典类型

temp_code_to_compare=self.params_interface['code_to_compare']# 获 取 待 比 较
code 名称
                if temp_code_to_compare in temp_result_interface.keys():
                    #if unicode(str(temp_result_interface
[temp_code_to_compare]),
"utf-8")==unicode(str(self.params_interface['code_expect']), "utf-8"):
                    if
temp_result_interface[temp_code_to_compare]==self.params_interface['code_
expect']:
                        result={'code':'0000','message':'关键字参数值相同
','data':[]}
                        operation_db.op_sql("UPDATE case_interface set
code_actual='%s',result_code_compare=%s where id=%s"
                                            %(temp_result_interface[temp_code_to
_compare],1,self.id_case))
                        #operation_db.OpSql("UPDATE case_interface set
code_actual=%s,result_code_compare=%s,result_interface='%s' where id=%s" %
```

```
(temp_result_interface[temp_code_to_compare], 0, result_interface,
self.id_case))
                elif
unicode(str(temp_result_interface[temp_code_to_compare]),
"utf-8")!=unicode(str(self.params_interface['code_expect']), "utf-8"):
                    result={'code':'1003','message':'关键字参数值不相同
','data':[]}
                    operation_db.op_sql("UPDATE case_interface set
code_actual='%s',result_code_compare=%s where id=%s"
                                        %(temp_result_interface[temp_code_to
_compare],0,self.id_case))
                else:
                    result={'code':'1002','message':'关键字参数值比较出错
','data':[]}
                    operation_db.op_sql("UPDATE case_interface set
code_actual='%s',result_code_compare=%s where id=%s"
                                        %(temp_result_interface[temp_code_to
_compare],3,self.id_case))
            else:
                result={'code':'1001','message':'返回包数据无关键字参数
','data':[]}
                operation_db.op_sql("UPDATE case_interface set
result_code_compare=%s where id=%s" %(2,self.id_case))
        else:
            result={'code':'1000','message':'返回包格式不合法','data':[]}
            operation_db.op_sql("UPDATE case_interface set
result_code_compare=%s where id=%s" %(4,self.id_case))
    except Exception as error:#记录日志到 log.txt 文件
        result={'code':'9999','message':'关键字参数值比较异常','data':[]}
        operation_db.op_sql("UPDATE case_interface set
result_code_compare=%s where id=%s" %(9,self.id_case))
        logging.basicConfig(filename = config.src_path +
'/log/syserror.log',level =
logging.DEBUG,format='%(asctime)s %(filename)s[line:%(lineno)d] %(levelna
me)s %(message)s')
        logger = logging.getLogger(_ _name_ _)
        logger.exception(error)
```

```
        return result
    #定义将接口返回数据中参数名写入列表中
    def get_compare_params(self,result_interface):
        '''
        :param result_interface: HTTP 返回包数据
        :return:返回码 code，返回信息 message，数据 data
        '''
        try:
            if result_interface.startswith('{') and isinstance
(result_interface,str):
                temp_result_interface=json.loads(result_interface)
                self.result_list_response=temp_result_interface.keys()
                result={'code':'0000','message':'成功','data':self.
result_list_response}
            else:
                result={'code':'1000','message':'返回包格式不合法','data':[]}
        except Exception as error:#记录日志到 log.txt 文件
            result={'code':'9999','message':'处理数据异常','data':[]}
            logging.basicConfig(filename = config.src_path + '/log/syserror.
log',level =
logging.DEBUG,format='%(asctime)s %(filename)s[line:%(lineno)d] %(levelna
me)s %(message)s')
            logger = logging.getLogger(_ _name_ _)
            logger.exception(error)
        return result
    #参数完整性比较方法，传参值与_ _recur_params 方法返回结果比较
    def compare_params_complete(self,result_interface):
        '''
        :param result_interface:接口 http 返回包
        :return:返回码 code，返回信息 message，数据 data
        '''
        try:
            temp_compare_params=self._ _recur_params(result_interface)#获取
返回包参数集
            if temp_compare_params['code']=='0000':
                temp_result_list_response=temp_compare_params['data']#获取接
口返回参数去重列表
```

```
            if self.params_to_compare==u'' or
isinstance(self.params_to_compare,(tuple,dict)):#判断用例中数据为空或类型不符
合
                result={'code':'4001','message':'用例中待比较参数集错误
','data':self.params_to_compare}
            else:
                list_params_to_compare=eval(self.params_to_compare)#将数
据库表 unicode 编码数据转换成原列表
                if
set(list_params_to_compare).issubset(set(temp_result_list_response)):#集合
的包含关系
                    result={'code':'0000','message':'参数完整性比较一致
','data':[]}
                    operation_db.op_sql('UPDATE case_interface set
params_actual="%s",result_params_compare=%s where
id="%s"'%(temp_result_list_response,1,self.id_case))
                else:
                    result={'code':'3001','message':'实际结果中元素不都在预
期结果中','data':[]}
                    operation_db.op_sql('UPDATE case_interface set
params_actual="%s",result_params_compare=%s where
id="%s"'%(temp_result_list_response,0,self.id_case))
          else:
              result={'code':'2001','message':'调用__recur_params 方法返回错
误','data':[]}
              operation_db.op_sql('UPDATE case_interface set
result_params_compare=%s where and id="%s"'%(2,self.id_case))
       except Exception as error:#记录日志到 log.txt 文件
          result={'code':'9999','message':'参数完整性比较异常','data':[]}
          operation_db.op_sql('UPDATE case_interface set
result_params_compare=%s where id="%s"'
                              %(9,self.id_case))
          logging.basicConfig(filename = config.src_path +
'/log/syserror.log',level =
logging.DEBUG,format='%(asctime)s %(filename)s[line:%(lineno)d] %(levelna
me)s %(message)s')
          logger = logging.getLogger(_ _name_ _)
```

```
            logger.exception(error)
        return result
    #定义递归方法
    def _ _recur_params(self,result_interface):
        #定义递归操作，将接口返回数据中参数名写入列表中(去重)
        try:
            if result_interface.startswith('{') and isinstance
(result_interface,str):#入参是字符串类型且能被转换成字典
                temp_result_interface=json.loads(result_interface)
                self.__recur_params(temp_result_interface)
            elif isinstance(result_interface, dict):  # 入参是字典
                for param, value in result_interface.iteritems():
                    self.result_list_response.append(param)
                    if isinstance(value,list):
                        for param in value:
                            self.__recur_params(param)
                    elif isinstance(value,dict):
                        self._ _recur_params(value)
                    else:
                        continue
            else:
                pass
        except Exception as error:#记录日志到 log.txt 文件
            logging.basicConfig(filename = config.src_path + '/log/syserror.
log',level                                                              =
logging.DEBUG,format='%(asctime)s %(filename)s[line:%(lineno)d] %(levelna
me)s %(message)s')
            logger = logging.getLogger(_ _name_ _)
            logger.exception(error)
            return {'code': '9999', 'message':'处理数据异常', 'data': []}
        return {'code':'0000','message':'成功
','data':list(set(self.result_list_response))}
 #测试
 if _ _name_ _ == "_ _main_ _":
    sen_sql="select * from case_interface where name_interface='getIpInfo.
php' and  id=1"
    params_interface=operation_db.select_one(sen_sql)
```

```
    result_interface=params_interface['data']['result_interface']
    test_compare_param=CompareParam(params_interface['data'])

result_compare_code=test_compare_param.compare_code(result_interface)#关键参数值比较
    print(result_compare_code)

   result_compare_params_complete=test_compare_param.compare_params_complete(result_interface)#参数完整性比较
 print(result_compare_params_complete)
```

代码：analyse.py

```
# -*- coding:utf-8 -*-
#_ _author_ _ = '大婶 N72'
'''
定义导出数据库表数据到 Excel 中
1.init 方法初始化获取配置文件数据
2.export2excel 为主方法
'''
from xlrd import open_workbook
from xlutils.copy import copy
import os,opmysql,logging,datetime
from public import config
operation_db=opmysql.OperationDbInterface(link_type=1)#实例化自动化测试数据库操作类
class AnalyseData(object):
    '''
    定义对接口测试数据分析的类，包含的方法有：
    1.导出测试数据到 Excel 中
    '''
    def _ _init_ _(self):
        self.field= config.field_excel#初始化配置文件
    #定义导出指定数据到 Excel 中
    def export2excel(self,names_export):
        '''
        :param names_export: 待导出的接口名称，列表形式数据
        :return:
```

```
        '''
        counts_export=len(names_export)#导出总数
        fail_export=[]#导出失败接口名列表
        try:
            src = open_workbook(config.src_path + '/report/report_module.xls',
formatting_info=True)
            destination = copy(src)
            dt=datetime.datetime.now().strftime("%Y%m%d%H%M%S")#当前时间戳
            filepath=config.src_path+'/report/'+str(dt)+'.xls'
            destination.save(filepath)#保存模板表格到新的目录下
            for name_interface in names_export:
                cases_interface=operation_db.select_all("select * from
case_interface where case_status=1 and name_interface='%s'"
                            %(name_interface))#获取指定接口的测试用例数据
                if   len(cases_interface['data'])!=0   and   cases_interface
['code']=='0000':
                    src = open_workbook(filepath,formatting_info=True)
                    destination = copy(src)
                    sheet                                                    =
destination.add_sheet(name_interface,cell_overwrite_ok=True)
                    for col in range(0,len(self.field)):
                        sheet.write(0,col,self.field[col])# 获取并写入数据段信息
到 sheet 中
                    for row in range(1,len(cases_interface['data'])+1):
                        for col in range(0,len(self.field)):
                            sheet.write(row,col,'%s'%cases_interface['data']
[row-1][col])#写数据到对应 excel 表中
                    destination.save(filepath)
                elif  len(cases_interface['data'])==0  and  cases_interface
['code']=='0000':
                    fail_export.append(name_interface)
                else:
                    fail_export.append(name_interface)
            result={'code':'0000','message':'导出总数：%s，失败数：%s'
                                        %(counts_export,len(fail_export)),'d
ata':fail_export}
        except Exception as error:#记录日志到 log.txt 文件
```

```
            result={'code':'9999','message':'导出过程异常|导出总数：%s，失败
数：%s'
                    %(counts_export,len(fail_export)),'data':fail_export}
            logging.basicConfig(filename = config.src_path +
'/log/syserror.log',level =
logging.DEBUG,format='%(asctime)s %(filename)s[line:%(lineno)d] %(levelna
me)s %(message)s')
            logger = logging.getLogger(_ _name_ _)
            logger.exception(error)
        return result
 if _ _name_ _ == "_ _main_ _":
    names_export=operation_db.select_one("select value_config from
config_total where status=1 and key_config='name_export'")#获取导出的接口数据
元组
    if names_export['code']=='0000':#判断查询结果
        temp_export=eval(names_export['data'][0])#获取查询数据，并将其转换成字典
        test_analyse_data=AnalyseData()
        result_export=test_analyse_data.export2excel(temp_export)#导出结果
        print (result_export)
    else:
        print('获取导出接口集失败')
```

代码：config.py

```
# -*- coding:utf-8 -*-
#_ _author_ _ = '大婶 N72'
import os
field_excel=['编号','接口名称','用例级别','请求类型','接口地址','接口头文件','接
口请求参数','接口返回包','待比较参数','实际参数值','预期参数值','参数值比较结果','待
比较参数集合','实际参数集合','参数完整性结果','用例状态','创建时间','更新时间']#导出
的 excel 表格标题
src_path = os.path.dirname(os.path.dirname(os.path.realpath(_ _file_ _)))#
当前代码所在目录的上级目录
```

代码：main.py

```
# -*- coding:utf-8 -*-
#_ _author_ _ = '大婶 N72'
'''
主代码，用于串联各个 py 文件
```

```
'''
import logging,os,re
from common import request, opmysql, analyse, compare
from public import config
base_request= request.RequestInterface()#实例化 http 请求
base_operationdb_interface= opmysql.OperationDbInterface()#实例化接口测试数
据库操作类
try:
    print("开始接口自动化程序，请选择操作类型(0|执行用例；1|导出测试结果)")
    value_input = raw_input('请输入操作类型:'.decode('utf-8').encode('gbk'))
    a=re.search(r'^[0-1]$',str(value_input))
    while not re.search(r'^[0-1]$',str(value_input)):#正则判断不在 0~1 的数字
        print("请输入正确的操作类型(0|执行用例；1|导出测试结果)")
        value_input = raw_input('请输入操作类型:'.decode
('utf-8').encode('gbk'))
    else:
        if value_input=='0':
            print("您输入的是：0|准备：执行测试用例")
            module_execute=base_operationdb_interface.select_all("SELECT
value_config from config_total where key_config='exe_setup' and status=1")#
获取待执行接口数据
            if  len(module_execute['data'])!=0  and  module_execute['code']
=='0000':
                for module_execute_one in module_execute['data']:

temp_module_execute=eval(module_execute_one['value_config'])#每个接口的字典
数据
                    for temp_name_interface,condition in
temp_module_execute.iteritems():
                        print('############开始执行接口：%s############\n' %
(temp_name_interface))
                        temp_level_check=condition['level_check']#检查级别
                        temp_level_exe=tuple(condition['level_exe'])#执行级别

data_case_interface=base_operationdb_interface .select_all("select * from
case_interface where case_status=1 and name_interface='%s' and exe_level
in %s" %(temp_name_interface,temp_level_exe))#获取接口测试数据
                        if data_case_interface['code']=='0000' and
```

```
len(data_case_interface['data'])!=0:
                        for temp_case_interface in data_case_interface
['data']:
              id_case=str(temp_case_interface['id'])#用例编号
              url_interface=temp_case_interface ['url_interface']#接口地址
              headerdata=eval(temp_case_interface ['header_interface'])#请
求头文件
              param_interface=temp_case_interface  ['params_interface']#接
口请求参数
              type_interface=temp_case_interface ['exe_mode']#执行环境
                           result_http_response=base_request.http_request
(interface_url=url_interface,headerdata=headerdata,interface_param=param_
interface,request_type=type_interface)#发送 http 请求
                           print("接口地址:%s\n 请求参数:%s\n 返回包数
据:%s" %(url_interface,param_interface,result_http_response))
                           base_operationdb_interface.op_sql("UPDATE
case_interface set result_interface='%s' where id=%s" %(result_http_response
['data'],id_case))#接口返回包写入用例表
                           if  result_http_response['code']=='0000'  and
len(result_http_response['data'])!=0:
                for child_level_check in temp_level_check:#循环检查级别
                base_compare= compare.CompareParam(temp_case_interface)#实
例化参数比较类
                if child_level_check in (0,u'0'):#执行关键参数值检查

result_compare_code=base_compare.compare_code   你  (result_http_response
['data'])
                print('用例编号：%s|检查级别：关键参数值|接口名称：%s|提示信
息：%s\n' %(id_case,temp_name_interface, result_compare_code['message']))
                elif child_level_check in (1,u'1'):#执行参数完整性检查
                result_compare_params_complete=
base_compare.compare_params_complete(result_http_response['data'])
                print('用例编号：%s|检查级别：关键参数值|接口名称：%s|提示信
息：%s\n' %(id_case,temp_name_interface,
result_compare_params_complete['message']))
                elif child_level_check in (2,u'2'):#执行功能测试，待开发
                                    pass
                elif child_level_check in (3,u'3'):#执行结构完整性检查，待开发
```

```
                                    pass
                                else:
                                    print('用例编号：%s|接口名称：%s|检查级别
错误：%s\n' %(id_case,temp_name_interface,child_level_check))
                            elif len(result_http_response['data'])==0:
                                print('用例编号：%s|接口名称：%s|错误信息：接口返
回数据为空\n' %(id_case,temp_name_interface))
                            else:
                                print('用例编号：%s|接口名称：%s|错误信
息：%s\n' %(id_case,temp_name_interface,result_http_response['message']))
                        elif len(data_case_interface['data'])==0:
                            print('接口名称：%s|错误信息：获取用例数据为空，请检查用例
\n' %(temp_name_interface))
                        else:
                            print('接口名称：%s|错误信息：获取用例数据失败|错误信
息：%s\n' %(temp_name_interface,data_case_interface['message']))
                        print(u'############结束执行接口：%s############\n' %
(temp_name_interface))
                else:
                    print('错误信息：待执行接口获取失败|错误信息：%s' %module_execute
['message'])
            elif value_input=='1':
                print("您输入的是：1|准备：导出测试结果，请注意查看目录：%s" %(config.
src_path+'\\report'))
                names_export=base_operationdb_interface.select_one("select
value_config from config_total where status=1 and key_config='name_export'")#
获取导出的接口数据元组
                if names_export['code']=='0000' and len(names_export['data']
['value_config'])!=0:#判断查询结果
                    temp_export=eval(names_export['data']['value_config'])# 获取
查询数据，并将其转换成字典
                    test_analyse_data= analyse.AnalyseData()#实例化数据分析类
                    result_export=test_analyse_data.export2excel(temp_export)#
导出结果
                    print (result_export['message'])
                    print("导出失败接口列表：%s\n" %result_export['data'])
                else:
                    print("请检查配置表数据正确性，当前值为：%s\n" %names_export
```

```
['data'])
 except Exception as error:#记录日志到 log.txt 文件
    print("系统出现异常：%s" %error)
    logging.basicConfig(filename = config.src_path + '\log\syserror.log',
level = logging.DEBUG,format='%(asctime)s %(filename)s[line:%(lineno)d] %
(levelname)s %(message)s')
    logger = logging.getLogger(_ _name_ _)
    logger.exception(error)
 raw_input('Press Enter to exit...')
```